Computational Science and Engineering
(Int. Master's Program)

Technische Universität München

Master's Thesis

Conjugate Heat Transfer with the Multiphysics Coupling Library preCICE

Author:	Lucía Cheung Yau
1st Examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
2nd Examiner:	Univ.-Prof. Dr. rer. nat. habil. Miriam Mehl
Assistant Advisor:	Dipl.-Math. Benjamin Uekermann
Submission Date:	December 15, 2016

Declaration

I confirm that this master's thesis is my own work and I have documented all sources and material used.

.....	.....
Date	Lucía Cheung Yau

Abstract

Conjugate heat transfer refers to the coupled analysis of the thermal interactions between fluids and solids. Earlier methods relied on an empirical constant, the heat transfer coefficient, which lumps together all the unknown information regarding the heat transfer process. Conjugation consists in solving the temperature and heat flux distributions at the fluid-solid interfaces as a coupled problem, without assuming a heat transfer coefficient. The purpose of this thesis is to implement a partitioned approach to perform conjugate heat transfer analysis, where separate fluid and solid solvers operate on their own domains, and the interface values are exchanged and solved for in an iterative way. The coupling is done with the multiphysics coupling library preCICE. The tasks mainly involve the implementation of the adapter codes, which are in charge of steering the respective fluid and solid solvers, and exchanging coupling data by calls to the preCICE library. The implementation is validated and applications are presented.

Acknowledgements

I would like to express my most sincere gratitude to all those who have made this thesis possible. Firstly, I would like to thank SimScale GmbH for giving me the opportunity to write this thesis and for providing me with all the support and the means I have needed to undertake this research. I would like to especially thank Babak Gholami, my supervisor at SimScale, for his day-to-day support, for the valuable discussions, and the guidance and encouragement provided.

I would also like to express my deepest gratitude and appreciation to my thesis advisor, Benjamin Uekermann, for the guidance, support and advices provided throughout the project. Thank you for always being ready to help and answer all my questions.

My deepest gratitude also goes to the National Secretariat of Science, Technology and Innovation of Panama (SENACYT) and the German Academic Exchange Service (DAAD), who have given me the opportunity and have provided me with the financial support to carry out my Master's studies in Germany.

Last but not least, I would like to thank my friends and family, especially my parents, for their unconditional love and support.

Contents

Chapter 1

Introduction

Conjugate heat transfer refers to the coupled analysis of heat transfer between fluids and solids. Heat transfer in fluids occurs through *convection*, which is a combination of two distinct mechanisms: the microscopic random motion of fluid molecules (diffusion) and the macroscopic bulk fluid motion (advection). In solids, only diffusion is possible (although *conduction* is the term used to refer to this phenomenon in solids), and it simply amounts to the vibration and collision of particles. Different partial differential equations (PDEs) are used to describe the different physics in the fluids and in the solids. These PDEs are coupled at the fluid-solid interfaces, where the continuity of the temperature and the heat flux must be satisfied.

Conjugate heat transfer is relevant in many engineering applications, where an accurate prediction of the heat transfer and the thermal loading is important. This includes the design of industrial machines and devices, where a proper thermal design has a direct impact on the performance and the lifetime of the devices. Some example applications include:

- Heating, ventilation, air-conditioning (HVAC)
- Cooling of electronic devices
- Heat exchangers
- Gas turbines
- Engine cooling

The aim of this thesis is to implement a numerical solver for conjugate heat transfer problems by coupling independent fluid and solid solvers (single-physics solvers). The coupling of single-physics solvers for treating multi-physics problems is referred to as partitioned approach. Instead of putting all the equations into one single system, the fluid and solid equations are solved separately by the single-physics solvers. The coupling is done externally, through the exchange of boundary values at the fluid-solid interfaces. This allows to reduce the time and effort required to build new solvers that deal with multiple physics.

Certainly, there are advantages and disadvantages of using a partitioned approach. The major disadvantage or challenge is having to deal with stability and convergence issues. Nevertheless, the flexibility that this approach offers, often outweighs its downside. A partitioned

approach allows to freely choose and reuse the most suitable tools at hand. These include the already mentioned single-physics solvers, as well as other tools such as meshing and post-processing utilities, that are used together with the solvers. Furthermore, it provides more flexibility in terms of allowing different length and time scales.

For this thesis, well-tested and well-validated open-source solvers are used. For solving the fluid flow and the heat convection, OpenFOAM is used. CalculiX and Code_Aster are used for solving the heat conduction in solids. The coupling is done through the coupling library preCICE [1], developed and maintained by the Technical University of Munich and the University of Stuttgart.

The coupling of different solvers with preCICE involves writing some "glue" code to be added to the solvers. This code that connects the solver with the preCICE library is called the coupling *adapter*. In very general terms, the adapter has to access and manipulate mesh and boundary data of the solver, control the time-stepping, and trigger the exchange of data through appropriate calls to the coupling library. This thesis documents the implementation and validation of the adapters, and demonstrates their use on relevant application scenarios.

The development of the thesis is carried out at SimScale GmbH, a company that develops cloud-based computer-aided engineering (CAE) software. The SimScale platform supports a variety of simulation types, including structural, fluid, thermodynamic, particle and acoustic simulations. The backend of the platform is based entirely on open-source solvers, including the ones used in this thesis, among others. Conjugate heat transfer simulations too are already possible on the SimScale platform. These are run with OpenFOAM CHT solvers. The intended reach of this thesis is not limited to obtaining a preCICE-coupled CHT solver, but to serve as a foundation for future work that further integrates these open-source solvers together. The flexibility of a partitioned approach and the features that preCICE offers, together make it an attractive way of expanding the number of analysis types that are possible on this cloud-based CAE platform.

One last remark worth mentioning is that preCICE has often been used for mechanical fluid-structure interaction (FSI) and acoustic coupling, but little has been done regarding thermal coupling. Therefore, this thesis also contributes to the development of this library, through the addition of a new multi-physics application and new adapters.

Thesis Structure

The content of this document is organized in the following way: the physics and the governing equations of conjugate heat transfer are presented in Chapter 2. Following that, an overview of the coupling library preCICE is provided in Chapter 3. The main components and features of the library are explained. Chapter 4 introduces the solvers used in this thesis, and documents the implementation of the coupling adapters for these solvers. In Chapter 5, validation cases are presented and the results are discussed and analyzed. Chapter 6 presents various demonstration cases that show the capabilities of the coupling developed in this thesis. Conclusions and suggestions for future work are given in Chapter 7.

Chapter 2

Conjugate Heat Transfer

In this chapter, relevant concepts regarding conjugate heat transfer are laid out. First, an overview of the physics of heat transfer is provided. Next, the governing equations and boundary conditions are presented. The chapter concludes with a description of the procedure for coupling the solution of the heat transfer in fluids and solids.

2.1 Mechanisms of Heat Transfer

Heat transfer is the flow of thermal energy that occurs whenever there is a temperature gradient in a medium. The main mechanisms by which heat is transferred are *conduction*, *convection* and *radiation*. This thesis deals only with the first two forms.

2.1.1 Heat Conduction

Heat conduction happens at a molecular level, where energy is transferred from more energetic particles to less energetic particles. The rate of the heat transfer between two bodies is proportional to the temperature difference between them. The rate of heat transfer per unit area or heat flux is given by Fourier's law of heat conduction

$$\boldsymbol{q} = -k\nabla T \tag{2.1}$$

where

- $\boldsymbol{q}$ $[W/m^2]$ is the heat flux density,
- k $[W/(m \cdot K)]$ is the material's thermal conductivity, which may be temperature-dependent, and
- ∇T $[K/m]$ is the temperature gradient.

The negative sign indicates that heat is transferred in the direction of lower temperature.

2.1.2 Heat Convection

Convective heat transfer happens in fluids, both because of the microscopic motion of the constituting particles and because of the bulk fluid motion.

Depending on the main driving force of the fluid motion, one can distinguish between two forms of convection: natural convection and forced convection. Natural convection happens due to the thermal expansion of fluids. The density variations produce buoyancy forces that set the fluid in motion. Most commonly, natural convection happens due to the presence of the gravitational field, which causes warmer, less dense fluid to rise, and colder, denser fluid to sink. However, in general, it can be caused by any body force that is proportional to the density. On the other hand, forced convection happens when the motion of the fluid is caused by an external source, like a fan or a blower, and buoyancy effects are negligible. Mixed convection occurs when both natural and forced convection are present.

Newton's law of cooling is often used to model the cooling of an object by convective heat transfer. It states that the rate of heat loss (cooling) is proportional to the temperature difference between the body and its surrounding medium (typically a fluid). Thus, the rate of heat loss is given by

$$\frac{dQ}{dt} = hA(T_s(t) - T_\infty) \tag{2.2}$$

where Q $[J]$ is the thermal energy, T_s is the temperature of the solid body, T_∞ is the environment temperature, and A $[m^2]$ is the heat transfer surface area. The constant of proportionality h $[W/(m^2 \cdot K)]$ is the heat transfer coefficient.

2.2 Material Properties and Dimensionless Quantities

Material properties that are relevant to heat transfer analysis include:

- Thermal conductivity k $[W/(m \cdot K)]$ – measures of how good a material is at conducting heat (e.g. materials of low thermal conductivity are used as insulators)
- Specific heat capacity c_p $[J/(kg \cdot K)]$ – is the amount of heat required to raise the temperature of 1 kg of substance by 1 K
- Thermal diffusivity α $[m^2/s]$ – is a measure of the thermal inertia of a material

The relationship between the thermal conductivity, the thermal diffusivity and the specific heat capacity is given by

$$k = \alpha \rho c_p. \tag{2.3}$$

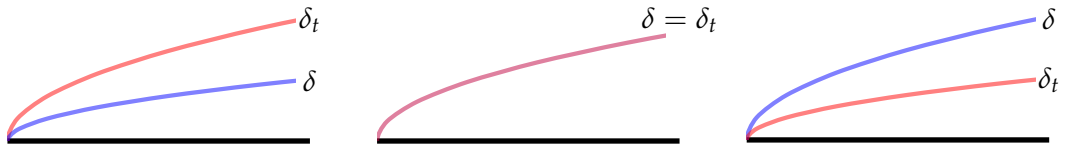
Dimensionless quantities that are important in heat transfer analysis include:

- The Prandtl number $Pr = \nu/\alpha$ – is the ratio of momentum to thermal diffusivity in a fluid. The effect of the Prandtl number on the thickness of the velocity and thermal boundary layers is shown in Figure 2.1.

- The turbulent Prandtl number $Pr_t = \nu_t/\alpha_t$ – is the ratio of the momentum eddy diffusivity to the heat transfer eddy diffusivity. It is important when solving heat transfer in turbulent boundary layer flows.
- The Biot number Bi – characterizes the intensity of the conjugate heat transfer [2]. It is defined as the ratio of the thermal resistance of the solid to that of the fluid, that is

$$Bi = \frac{hL}{k_s} \tag{2.4}$$

where h is the convective heat transfer coefficient, L is the characteristic length and k_s is the thermal conductivity of the solid.



(a) $Pr \ll 1$ (b) $Pr = 1$ (c) $Pr \gg 1$

Figure 2.1: Effect of the Prandtl number on the thickness of the velocity and thermal boundary layers (δ and δ_t, respectively)

2.3 Governing Equations and Boundary Conditions

In this section, the partial differential equations for modelling heat transfer in fluids and solids are presented.

2.3.1 Governing Equation for the Solid

Heat transfer in solids is governed by the heat conduction equation

$$\rho c_p \frac{\partial T}{\partial t} - \nabla \cdot (k \nabla T) = Q \tag{2.5}$$

where ρ is the density, c_p the specific heat capacity, k the thermal conductivity and Q is a volumetric heat source.

2.3.2 Governing Equations for the Fluid

The fluid flow is governed by the Navier-Stokes equations, which is a set of three coupled equations: the continuity equation, the momentum equation and the energy equation, respectively

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0, \tag{2.6}$$

$$\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \times \boldsymbol{u}) + \nabla p - \nabla \cdot \boldsymbol{\tau} = \rho \boldsymbol{g}, \tag{2.7}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} E) + \nabla \cdot (\boldsymbol{u} p) - \nabla \cdot (\boldsymbol{\tau} \cdot \boldsymbol{u}) - \nabla \cdot (k \nabla T) = \rho r + \rho \boldsymbol{g} \cdot \boldsymbol{u} \tag{2.8}$$

where

- ρ is the density
- $\boldsymbol{u}$ is the velocity
- p is the pressure
- $\boldsymbol{\tau}$ is the viscous stress tensor
- $\boldsymbol{g}$ is any body acceleration (such as gravity)
- E is the specific total energy
- T is the absolute temperature
- k is the thermal conductivity
- r is any specific heat source (e.g. per unit mass)

For incompressible flows, the density ρ is assumed to be constant, and the energy equation becomes decoupled. The continuity equation and the momentum equation can be solved first, and then the temperature distribution can be obtained by plugging-in the velocity and pressure values in the energy equation. However, for compressible flows it is necessary to solve the energy equation together with the continuity and the momentum equations.

The specific total energy E is the sum of the specific internal energy e and the specific kinetic energy $K = \|\boldsymbol{u}\|^2/2$, i.e. $E = e + K$. The energy equation can also be written in terms of the enthalpy, which is the sum of the internal energy e and the kinematic pressure $h = e + p/\rho$. The specific enthalpy is typically modelled as proportional to the temperature, with the specific heat capacity c_p as the constant of proportionality, thus providing a relationship between energy and temperature

$$h = c_p T. \tag{2.9}$$

In order to solve for all the unknowns (velocity, pressure, density, temperature), an additional equation is required. This is the *equation of state*, which defines the relation between density, pressure and temperature. It depends on the type of fluid being modelled. For example, for gases, the ideal gas law is typically used

$$p = \rho R T \tag{2.10}$$

where $R = 8.3144598\ J/(mol \cdot K)$ is the gas constant.

2.3.3 Boundary Conditions

The thermal boundary conditions that can be applied to both the fluid and the solid domains are explained in this section.

A **Dirichlet** boundary condition, also referred to as boundary condition of the first kind, prescribes a temperature at the boundary

$$T = T_D \text{ on } \Gamma_D. \tag{2.11}$$

A **Neumann** boundary condition, or boundary condition of the second kind, prescribes the heat flux at the boundary

$$k\frac{\partial T}{\partial \boldsymbol{n}} = q_N \text{ on } \Gamma_N \tag{2.12}$$

where $\partial T/\partial \boldsymbol{n}$ denotes the normal derivative at the boundary.

A special case of this boundary condition can be used to model perfectly insulated or *adiabatic* walls, by setting the surface normal temperature gradient to zero, reading

$$\frac{\partial T}{\partial \boldsymbol{n}} = 0 \text{ on } \Gamma_N. \tag{2.13}$$

A **Robin** boundary condition, or boundary condition of the third kind, is a weighted combination of a Dirichlet and a Neumann boundary condition

$$aT + b\frac{\partial T}{\partial \boldsymbol{n}} = g \text{ on } \Gamma_R. \tag{2.14}$$

In heat transfer problems, the Robin boundary condition typically appears as a *convective boundary condition*, with the form

$$k\frac{\partial T}{\partial \boldsymbol{n}} + h(T - T_\infty) = 0, \tag{2.15}$$

or more commonly,

$$q = h(T - T_\infty), \tag{2.16}$$

where q is the heat flux, h is the heat transfer coefficient and T_∞ is the sink (or environment) temperature.

In an uncoupled heat transfer analysis, these three types of boundary conditions are commonly used at the fluid-solid interfaces as approximations. For example, uniform temperature or uniform heat flux may be assumed. Convective boundary conditions are also often used, where the combined effects of the fluid flow are lumped together in the heat transfer coefficient h, which has to be obtained empirically. This coefficient is not a material property,

but depends on many factors such as geometry, surface temperature distribution, flow conditions, thermo-physical properties of the fluid, etc. For many applications, it might not be practical or feasible to obtain this parameter empirically.

Conjugate Boundary Condition

In conjugate heat transfer problems, boundary conditions of the fourth kind, or interface boundary conditions, state the continuity of the temperature and the heat flux at the fluid-solid interface Γ_{FS}. This reads

$$T_s = T_f \text{ on } \Gamma_{FS}, \tag{2.17}$$

and

$$k_s \frac{\partial T_s}{\partial \boldsymbol{n}} = -k_f \frac{\partial T_f}{\partial \boldsymbol{n}} \text{ on } \Gamma_{FS}, \tag{2.18}$$

with

$$\Gamma_{FS} = \Gamma_F \cap \Gamma_S. \tag{2.19}$$

No assumptions need to be made regarding the temperature and heat flux distributions at the interfaces, as they are obtained as part of the solution to the conjugate problem. A heat transfer coefficient is also no longer needed; instead, it can be obtained as a result of the coupled analysis.

2.4 Coupling Approaches

There are two major approaches for solving the coupled problem: the *monolithic* approach and the *partitioned* approach. In the monolithic approach, a global system of equation is obtained, involving both fluid and solid domains. The degrees of freedom of the fluid and the solid are solved together.

The following explanations are extracted from [1], where the coupling of one fluid solver and one solid solver is assumed. The monolithic system $\boldsymbol{A}$ involves the unknowns from the fluid domain $\boldsymbol{y_F}$, as well as the unknowns from the solid domain $\boldsymbol{y_S}$, which are all solved for at once. The monolithic system reads

$$\boldsymbol{A}(\boldsymbol{y_F}, \boldsymbol{y_S}) = 0. \tag{2.20}$$

In the partitioned approach, the equations are solved separately, and the coupling is done by exchanging the boundary values. This can be expressed as

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{s}) &= \boldsymbol{f} \\ \boldsymbol{S}(\boldsymbol{f}) &= \boldsymbol{s} \end{aligned} \tag{2.21}$$

where $\boldsymbol{F}$ is the fluid operator, $\boldsymbol{f}$ the output boundary values obtained from solving the fluid system, $\boldsymbol{S}$ is the solid operator and $\boldsymbol{s}$ is the output boundary values obtained from solving the solid system. Depending on the type of decomposition used, $\boldsymbol{s}$ and $\boldsymbol{f}$ may be different physical variables. For example, in a Dirichlet-Neumann decomposition of the problem, $\boldsymbol{s}$ may be the temperature computed at the solid side of the interface and used as a boundary condition for the fluid, and $\boldsymbol{f}$ the heat flux computed at the fluid side and used as a boundary condition on the solid side, or vice-versa.

The methods for solving the coupled equations (2.21) are explained in more detail in Chapter 3, where the different coupling schemes offered by the preCICE library are explained.

2.4.1 Coupling Boundary Conditions in a Partitioned Approach

In a partitioned approach, the solution of the coupled problem is obtained by exchanging the boundary values of the two sub-domains and iterating the equations (2.21). In this thesis, Dirichlet-Neumann and Robin-Robin partitioning of the system are considered. They present different behaviours in terms of stability and convergence, which are compared in Chapter 5, through a validation case.

Dirichlet-Neumann

In a Dirichlet-Neumann decomposition, temperatures and heat fluxes are exchanged. One domain has a temperature boundary condition and computes the heat flux at the conjugate boundary, whereas the other domain uses a heat flux boundary condition and in turn computes the temperature at the conjugate boundary.

The conjugate boundary conditions (2.17) and (2.18) are enforced by directly applying them as the boundary conditions of the individual domains.

A priori there is no difference between using a Dirichlet boundary condition on the fluid side and a Neumann boundary condition on the solid side or vice-versa. However, it is shown in [3] that there are differences in their stability and convergence behaviour.

Robin-Robin

When using a Robin-Robin decomposition, a heat transfer coefficient h and a sink temperature T_∞ are exchanged. Robin-Robin coupling boundary conditions are typically more robust, although more data has to be exchanged.

The coupling procedure with Robin boundary conditions is described below. The superscript i denotes the iteration. The subscript f and s denote fluid and solid, respectively. T_i and T_∞ are the interface temperature and the sink temperature, respectively. For simplification, the heat transfer coefficients h_s and h_f are assumed to be constant throughout the computation. The actual computation of the coupling variables T_∞ and h is explained in the Chapter 4, dedicated to implementation details. For further simplification, a Gauss-Seidel type of coupling is assumed, where one system is solved after the other. Other schemes are possible, and are explained in Section 3.1.

1. $i = 0$
2. Solve fluid with BC $q_f^i = h_s(T_{if}^i - T_{\infty s}^i)$
3. Compute sink temperatures $T_{\infty f}^i = T_{if}^i + q_f^i / h_f$
4. Send sink temperatures $T_{\infty f}^i$ from the fluid solver to the solid solver
5. Solve solid with BC $q_s^i = h_f(T_{is}^i - T_{\infty f}^i)$
6. Compute sink temperatures $T_{\infty s}^{i+1} = T_{is}^i + q_s^i / h_s$
7. Send sink temperatures $T_{\infty s}^{i+1}$ from the solid solver to the fluid solver
8. $i = i + 1$
9. Repeat from 2 until $|T_{\infty s}^{i+1} - T_{\infty s}^i| < \epsilon$

Listing 2.1: Coupling with Robin boundary conditions

From the first four equations in Listing 2.1, it can be shown that when the coupling converges, i.e. $|T_{\infty s}^{i+1} - T_{\infty s}^i| \to 0$, then also $|T_{if}^i - T_{is}^i| \to 0$ and $|q_f^i + q_s^i| \to 0$ hold true, which means that the conjugate boundary conditions (2.17) and (2.18) are satisfied.

According to [3], the value of the heat transfer coefficient h does not affect the final solution but only the convergence speed of the coupling. In [4] a stability analysis is performed, and an optimal coefficient is obtained for a model problem. The influence of the parameters h and T_∞ on the convergence speed has not been studied in this thesis.

Whether Robin boundary conditions can be used for the coupling with preCICE, depends on whether the specific solvers support Robin boundary conditions and whether the parameters can be externally modified.

Chapter 3

Overview of the Coupling Library preCICE

preCICE (*Precise Code Interaction Coupling Environment*)[1] is an open-source library for surface-coupled partitioned multiphysics simulations. The philosophy behind preCICE is to make it possible to take full advantage of existing single-physics simulation codes by reusing them for multi-physics simulations. This requires modifications to be performed on the solver code, in order to create a well-defined interface that allows this solver to be coupled, in a plug-and-play fashion, with other solvers that have also been adapted with preCICE. This type of coupling achieved with preCICE is also referred to as *black-box* coupling in the literature, given that the code changes necessary to couple existing solvers require minimal knowledge of their internal workings.

The main components of preCICE, which allow the interaction of two or more solver codes to obtain the solution of the multi-physics problem are:

- Equation coupling,
- Data mapping, and
- Communication.

The code that needs to be integrated into the single-physics solvers in order for them to use preCICE is generally referred to as the coupling *adapter*. In other words, the adapter connects the solver with preCICE, which in turn handles the communication and data exchange with other solvers. Code modifications to the original solvers should be kept to a minimum, and should be as non-invasive as possible. The individual components of preCICE are fully configured through an XML file, which is loaded at runtime. Therefore, the particular choices for the equation coupling schemes, data-mapping and communication, do not affect the implementation of the adapter.

In the rest of the document, the terms *participant* and *solver* may be used interchangeably. The coupling of more than two participants is also possible with preCICE.

[1] `http://www.precice.org/`

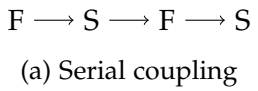
A brief review of the components that are most relevant to this thesis, namely the equation coupling and the data mapping components, will be provided in the following sections. Regarding the communication, preCICE provides both TCP/IP socket-based as well as MPI-based communication methods between the solver executables. The library provides a vast array of options for each component, such that a complete and detailed documentation exceeds the scope of this thesis. Therefore the reader is referred to the original documentation of preCICE [1] for a much more complete description of its features and the implementation details. To finalize this chapter, a section is also dedicated to the application programming interface (API) of preCICE, which has to be introduced in the single-physics solver codes.

3.1 Equation Coupling

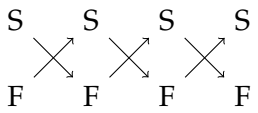
The equation coupling schemes determine how the solutions from the individual solvers are combined. In this respect, preCICE offers a variety of different schemes, which can be classified as implicit or explicit, serial or parallel. The four possible combinations are available to use in preCICE (serial-explicit, serial-implicit, parallel-implicit and parallel-explicit).

In the case of **serial coupling**, each solver takes turns to solve. The first participant uses data from the previous iteration of the second participant to compute its solution for the current iteration. It then sends the data to the second participant, which computes its solution also for the current iteration. In the case of **parallel coupling**, both solvers solve at the same time, using data from the other solver from the previous iteration. This is represented schematically in Figure 3.1. Note that *previous iteration* here refers to the previous "solve", which can belong to the same or to different time steps.

The kind of parallelism described here is referred to as *interfield* parallelism, which is different from *intrafield* parallelism, where the degrees of freedom within one solver are solved in parallel. preCICE supports interfield parallelism, and also intrafield parallelism, as long as it is supported by the solver.



(a) Serial coupling



(b) Parallel coupling

Figure 3.1: Serial vs parallel coupling (image adapted from [5])

Explicit Coupling

Explicit coupling consists in solving the individual systems and exchanging the coupling data *a fixed number of times* per time step, regardless of the convergence of the coupling. There are various types of explicit coupling schemes. In preCICE, the *conventional serial staggered scheme*

(CSS) and the *conventional parallel staggered* (CPS) schemes [6] are available. In both of these schemes, the coupling data is exchanged only *once* per time step. The algorithms of the CSS and CPS schemes are shown in Listing 3.1 and 3.2, respectively.

```
for n = 0 to n_end:
    solve F^n(s^n) = f^{n+1}
    solve S^n(f^{n+1}) = s^{n+1}
```

Listing 3.1: Serial Explicit: Conventional Serial Staggered (CSS) Scheme [6]

```
for n = 0 to n_end
    solve F^n(s^n) = f^{n+1}
    solve S^n(f^n) = s^{n+1}
```

Listing 3.2: Parallel Explicit: Conventional Parallel Staggered (CPS) Scheme [6]

With an explicit coupling scheme, the monolithic system (2.20) is only approximately solved. In the context of CHT problems, this means that the continuity of the temperature and heat flux at the interfaces is not guaranteed. Furthermore, explicit schemes often lead to numerical instabilities. In such cases, implicit coupling becomes necessary.

Implicit Coupling

Implicit coupling schemes try to recover the solution of the monolithic system iteratively. Coupling iterations are performed until a prescribed convergence tolerance is reached. Performing multiple coupling iterations per time step might lead to the same instabilities present in the explicit schemes, therefore, implicit schemes are generally used together with techniques that stabilize the coupling and accelerate the convergence. These are called data post-processing techniques, since they modify the coupling data that is generated by the solvers.

Implicit coupling schemes reformulate equations (2.21) as a fixed-point problem

$$\boldsymbol{S} \circ \boldsymbol{F}(\boldsymbol{s}) = \boldsymbol{s}, \tag{3.1}$$

for which one iteration can be written as

$$\tilde{\boldsymbol{s}}_{k+1} = \boldsymbol{S}(\boldsymbol{F}(\boldsymbol{s}_k)) = \boldsymbol{S} \circ \boldsymbol{F}(\boldsymbol{s}_k). \tag{3.2}$$

Here, $\tilde{\boldsymbol{s}}_{k+1}$ denotes the values before post-processing, and $\boldsymbol{s}_{k+1}$, after post-processing.

The residual of each iteration is defined as

$$\boldsymbol{r}_{k+1} = \boldsymbol{S} \circ \boldsymbol{F}(\boldsymbol{s}_k) - \boldsymbol{s}_k = 0 \tag{3.3}$$

and becomes zero when the fixed-point problem is solved, reading

$$\boldsymbol{r} = \boldsymbol{S} \circ \boldsymbol{F}(\boldsymbol{s}) - \boldsymbol{s} = 0. \tag{3.4}$$

For practical purposes, the norm of the residual is monitored to determine convergence. Different convergence criteria can be defined. An absolute convergence measure reads

$$\|\boldsymbol{r}_{k+1}\|_{L^2} < \epsilon_{abs} \tag{3.5}$$

and a relative convergence measure reads

$$\frac{\|\boldsymbol{r}_{k+1}\|_{L^2}}{\|\tilde{\boldsymbol{s}}_{k+1}\|_{L^2}} < \epsilon_{rel}. \tag{3.6}$$

The methods for solving the fixed-point problem can be classified into two categories:

- **Schwarz procedures**: This group of methods simply iterate the Equations 2.21 to solve the fixed-point problem. Equations 2.21 can be solved either sequentially or in parallel. The serial version of the algorithm is also known as *multiplicative Schwarz method*, and the parallel version, as *additive Schwarz method*. Relaxation is used as post-processing for stabilizing the iterations. Relaxation consists in linearly combining the previous solution with the current (unrelaxed) solution, written as

  $$\boldsymbol{s}_{k+1} = (1 - \omega_k)\boldsymbol{s}_k + \omega_k \tilde{\boldsymbol{s}}_{k+1} \tag{3.7}$$

  Available relaxation methods in preCICE include:

  - *Constant underrelaxation*: The relaxation factor is constant, $\omega_k = \omega$.
  - *Aitken-based dynamic relaxation*: Convergence can be improved by using an optimal relaxation parameter, computed using Aitken's method, instead of using a constant relaxation parameter. Two previous iterates are used to compute the relaxation factor at the k-th iteration. For scalar iterations, the relaxation can be written as

    $$\omega_k = -\omega_{k-1} \frac{s_{k-1} - s_k}{(s_{k-1} - \tilde{s}_k) - (s_k - \tilde{s}_{k+1})} = -\omega_{k-1} \frac{r_k}{r_{k+1} - r_k}. \tag{3.8}$$

    For vector iterations, it can be written as

    $$\omega_k = -\omega_{k-1} \frac{\boldsymbol{r}_k^T (\boldsymbol{r}_{k+1} - \boldsymbol{r}_k)}{\|\boldsymbol{r}_{k+1} - \boldsymbol{r}_k\|_{L_2}^2}. \tag{3.9}$$

- **Newton's method**: Newton's method is often used to solve fixed-point problems such as (3.1). One Newton iteration reads

  $$\boldsymbol{s}_{k+1} = \boldsymbol{s}_k - \boldsymbol{J}_{\boldsymbol{R}}^{-1}(\boldsymbol{s}_k)\boldsymbol{R}(\boldsymbol{s}_k) \tag{3.10}$$

  where

  $$\boldsymbol{R}(\boldsymbol{s}) = \boldsymbol{S} \circ \boldsymbol{F}(\boldsymbol{s}) - \boldsymbol{s} \tag{3.11}$$

and $\boldsymbol{J_R}$ is the Jacobian of $\boldsymbol{R}$. The main question here is how to obtain the Jacobian, which is not available in the black-box solvers. There are various methods which differ in how they approximate the Jacobian. The Interface Quasi-Newton Method (*IQN-ILS*) method, often used in FSI problems, uses a least-squares approximation of the inverse of the Jacobian. The complete derivation of the IQN-ILS method falls outside the scope of this thesis, and therefore, the reader is referred to a complete description in [7] and [1]. preCICE implements various convergence acceleration schemes that are based on Newton's method. Again, for a complete description the reader is referred to the original documentation of preCICE.

3.2 Data Mapping

In partitioned approaches, individual domains are typically discretized independently, leading to meshes that do not match at the interfaces. preCICE offers multiple data-mapping methods, which allow transferring data between non-matching surface meshes. The mapping methods include projection-based mapping and radial basis function interpolation.

Data mapping methods can be either *consistent* or *conservative*. Consistent mapping implies that a constant function is mapped exactly, while conservative mapping means that the sum of mapped values are the same on both sides. Both types of mapping are available in preCICE; the correct choice depends on the coupling variables used. Consistent mapping is typically used for variables such as positions, fluxes and densities, whereas conservative mapping is used for integral values [8]. In this thesis, the coupling variables are temperature, (distributed) heat flux and heat transfer coefficient. For these variables, consistent mapping is the appropriate choice. Variables can be transformed to use a different type of mapping, but it must be carefully determined whether this brings any advantage. A comparison between consistent and conservative mapping approaches can be found in [9].

The data mapping can be done without extensive knowledge about the discretization used by the solvers or the shape functions used, as for most of the methods it is enough to provide a cloud of points with the associated coupling data.

In the following paragraphs, the mapping methods available in preCICE are very briefly described. The values are mapped from a *source mesh* to a *target mesh*. The explanations assume that *consistent* mapping is used, as it is the type of mapping relevant to this thesis.

Projection-Based Mapping

The simplest mapping method available in preCICE is the **nearest-neighbor** mapping. The node of the target mesh simply takes the value of the closest node from the source mesh. This is a first-order accurate method.

Another projection-based method available in preCICE, which is second-order accurate, is the **nearest-projection** method. To compute the mapped value, the node in the target mesh is first orthogonally projected onto the source mesh. Then, its value is obtained by linear interpolation from the node values on the source mesh. In order for this type of mapping to

work, in addition to the location of the nodes, it is also necessary to provide their connectivity (i.e. how the nodes are connected to make up the faces).

Radial-Basis Function Interpolation

This type of mapping constructs an interpolant with the nodes and data of the source mesh. The interpolant is then evaluated at the nodes of the target mesh. Radial basis functions (RBF) are used as the basis of the interpolant. An RBF is a function whose value depends only on the distance from an arbitrary point. The RBFs are enriched with a global first-order polynomial, in order to ensure the exact interpolation of constant and linear functions.

Several types of RBF-based mapping methods are implemented in preCICE. These include RBFs of global support, such as: Gaussian, multiquadrics, inverse multiquadrics, thin plate splines and volume splines; as well as RBFs of compact support: compact thin plate splines C2, compact polynomial C0 and compact polynomial C6. For the RBFs with compact support, a support radius must be specified, which limits the range of influence of a node. A wider support radius gives a better approximation, but is computationally more expensive, whereas a narrower support radius requires less computational effort but may lead to larger mapping errors.

3.3 Application Programming Interface of preCICE

In very general terms, a numerical solver typically consists of a loop that solves a system of partial differential equations. The loop is usually a time-stepping loop, but may also be a loop that solves the equations iteratively. Transient solvers often consist of a double loop, with an outer-loop for time stepping and an inner loop for solving the equations iteratively.

In order to couple the solver, the loop must be modified by introducing calls to the preCICE library. At the very least, boundary values need to be extracted, exchanged and updated, within the loop. For explicit coupling it is enough to expose the *data locations* and the *data values* to preCICE. However, to support more advanced coupling schemes and features, other changes are necessary. To support implicit coupling, *control of the time stepping* must be given to preCICE, because a time step usually has to be solved multiple times with updated coupling data until it converges.

There are usually two ways to allow the same time step to be solved more than once. One way is by storing the state of the solver (i.e. writing a checkpoint) right before solving a new time step. This way, if the coupling does not converge, the solver can be reverted to its previous state (i.e. reading the checkpoint), and solve again the same step with different coupling boundary data. A second way is possible if the solver uses iterations inside the time stepping loop. The solver iterations can be fused with the coupling iterations, and checkpointing can be avoided: each coupling iteration translates into performing more solver iterations, with updated boundary values. The solver advances in time only after the coupling has converged.

If sub-cycling is used, checkpointing is always necessary. Sub-cycling means that a solver may advance several time steps between every exchange of coupling data. The adapter must

```
void setMeshVertices(int meshID, int size, double * positions, int * ids);
void setMeshTriangleWithEdges(int meshID, int firstVertexID, int secondVertexID,
   int thirdVertexID);
int getMeshID (const std::string& meshName);
int getDataID(const std::string& dataName, int meshID);
double initialize();
void initializeData();
bool isCouplingOngoing();
double advance(double computedTimestepLength);
bool isReadDataAvailable();
bool isWriteDataRequired(double computedTimestepLength);
void readBlockScalarData(int dataID, int size, int * valueIndices,
   double * values);
void writeBlockScalarData(int dataID, int size, int * valueIndices,
   double * values);
bool isActionRequired(const std::string& action);
void fulfilledAction(const std::string& action);
```

Listing 3.3: Excerpt of preCICE's API

be able to *adjust the time step size* used by the solvers, to make sure that the sub-steps add up to the coupling time step size, so that they can synchronize.

Listing 3.3 shows part of the API provided by preCICE to carry out the operations explained above. These are methods of the `precice::SolverInterface` class.

Listing 3.4 shows the typical structure of an adapted solver. Before the solver loop starts, the location of the coupling data must be provided to preCICE and the coupling must be initialized, by calling the methods `setMeshVertices` and `initialize`, respectively. The control of the solver iterations or time stepping is handed to preCICE, by using the method `isCouplingOngoing` to determine whether more time steps or iterations need to be solved. Before solving the equations (represented by `solve()`), the coupling data is read with `readBlockScalarData` and the solver's boundary conditions are updated. After solving, the boundary values are extracted and written to preCICE's buffer with `writeBlockScalarData`. It is important to point out here that these methods do not send or receive data. The actual exchange of data is taken care of by the `advance` method. This is because additional logic and steps are involved in the coupling (such as data mapping, convergence acceleration and convergence measurement), beyond simply exchanging the data. When calling `advance`, the step size used by the solver must be passed to the function call, so that preCICE can determine whether the solvers have synchronized and coupling operations must be carried out. The `advance` method returns the maximum time step size that the solver can use in the next step. Additionally, as mentioned before, the state of the solver has to be stored at the beginning of a new *coupling time step*, and must be reloaded before a repeated coupling iteration. To determine whether these actions are necessary, the `isActionRequired` method is used. To notify preCICE that certain actions have been carried out, the method `fulfilledAction` is used.

```
for each interface:
   precice.setMeshVertices(...)

precice_dt = precice.initialize()

if precice.isActionRequired("write-initial-data"):
   for each interface:
      extractBoundaryValues()
      precice.writeScalarBlockData(...)
   precice.fulfilledAction("write-initial-data")

precice.initializeData()

while precice.isCouplingOngoing():

   if precice.isActionRequired("write-iteration-checkpoint"):
      writeCheckpoint()
      precice.fulfilledAction("write-iteration-checkpoint")

   if precice.isReadDataAvailable():
      for each interface:
         precice.readBlockScalarData(...)
         updateBoundaryValues()

   solver_dt = solver.determineStepSize()
   solver_dt = min(precice_dt, solver_dt)

   solver.solve(solver_dt)

   if precice.isWriteDataRequired(solver_dt):
      for each interface:
         extractBoundaryValues()
         precice.writeBlockScalarData(...)

   precice_dt = advance(solver_dt)

   if precice.isActionRequired("read-iteration-checkpoint"):
      readCheckpoint()
      precice.fulfilledAction("read-iteration-checkpoint")
```

Listing 3.4: Typical structure of an adapted solver

Chapter 4

Implementation of the Coupling Adapters

This chapter describes the implementation of the coupling adapters for conjugate heat transfer for three different open-source numerical simulation packages: OpenFOAM for fluid simulation, and CalculiX and Code_Aster for solid simulation. First, some general considerations that are relevant to the three solvers are presented. Subsequently, the coupling of each solver is described, providing relevant solver-specific details, which are organized in the following way:

- First, an overview of the approach used to couple the solver is provided, considering the general structure of the solver package.
- The adapter is then described, in terms of the files and functions that have been implemented to couple the solver.
- Modifications to be performed to the solver code in order to integrate it with the adapter are explained.
- The location of the coupling data is also described. The application of boundary conditions and the extraction of boundary values is explained.
- Finally, some details regarding steady-state simulations and parallelization are provided.

4.1 General Considerations

Before presenting the solver-specific implementation details, some general considerations regarding all three adapters are first presented.

4.1.1 Structure of the coupling adapter

The coupling adapters developed in this thesis basically have two responsibilities:

1. Grouping together several (high-level) coupling functionalities offered by preCICE, to minimize the amount of changes to the original solver; and
2. Dealing with lower level access and manipulation of the solver's data structures (e.g. solver-specific operations for updating boundary conditions).

The higher-level functions of the adapter are described below, where `precice` is an instance of `precice::SolverInterface`.

- **configure()/setup()/constructor**:
  - Creates the interfaces: Configures the surface meshes by extracting the location of the nodes and/or face centers (solver-specific) and calling `precice.setMeshVertices(...)` to expose the data locations to preCICE. It also configures the coupling data and queries for the data IDs with `precice.getDataID(...)`.
- **initialize()**: Groups various operations from preCICE:
  - Calls `precice.initialize()` to initialize the coupling,
  - Checks whether it is necessary to write the initial data for the other solver. If yes, then it extracts the boundary data and calls `precice.writeScalarBlockData(...)` for each interface;
  - Calls `precice.initializeData()` to trigger the exchange of initial data.
- **writeCheckpoint()**: Makes a copy of the solution fields and stores the current simulation time (solver-specific).
- **readCouplingData()**: Checks if there is read-data available with `precice.isReadDataAvailable()`. If yes, then for each interface, it will perform the following operations:
  - Read the coupling data from preCICE with `precice.readScalarBlockData(...)`,
  - Update the boundary values (solver-specific).
- **adjustSolverTimeStep()**: The time step size of the solver is adjusted. The minimum between the step size determined by the solver (e.g. based on stability constraints) and the step size determined by the coupling is taken.
- **writeCouplingData()**: After solving, if `precice.isWriteDataRequired(dt)` is true, these operations are carried out for each interface:
  - Extract boundary values (solver-specific),
  - Write the boundary values to preCICE with `precice.writeBlockScalarData(...)`.
- **advance()**: Calls `precice.advance()`. This is the most important coupling step, where the coupling data is exchanged. It involves data communication and mapping, data post-processing (e.g. convergence acceleration), convergence measurement.
- **readCheckpoint()**: Reloads the copies of the solution fields and resets the time (solver-specific).

Depending on the actual implementation of the adapters, some of the features of preCICE might or might not be available. For the intended purposes of the adapters of this thesis, it

is important that the adapter be able to handle an arbitrary number of interfaces. Also, the adapter must support different boundary conditions for the coupling (Dirichlet, Neumann and Robin), so that they can be tested and compared. Finally, transient and steady-state simulations must be supported.

4.1.2 Robin or Convective Boundary Conditions

In the case of Dirichlet and Neumann coupling boundary conditions, the meaning of the variables (temperature and heat flux) is clear. However, for Robin or convective boundary conditions, further explanation is required regarding how the heat transfer coefficient h and the sink temperature T_∞ are computed.

The following explanations refer to the Figure 4.1, where the coupling interface and two adjacent cells, one from each domain, are depicted. T_1 refers to the temperature at an interior point of Ω_1 located at a distance Δx_1 from the interface, and T_2 the temperature at an interior point of Ω_2, located at a distance Δx_2 from the interface. T_i is the temperature at the interface.

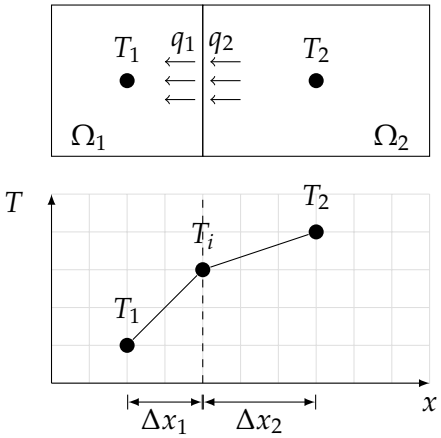


Figure 4.1: Sink temperature and heat transfer coefficient

To determine the value of h and T_∞ to be used, balance of fluxes at the boundary is applied and a first-order approximation of the derivatives is used, which reads

$$q_1 = -q_2 \approx k_2 \frac{T_i - T_2}{\Delta x_2} \tag{4.1}$$

where q_1 is the flux computed at the the left domain Ω_1, q_2 is the flux computed at the right domain Ω_2.

The boundary condition for Ω_1 can then be rewritten in the form of (2.16), as

$$q_1 = h_2(T - T_{\infty 2}) \tag{4.2}$$

with

$$\begin{aligned} h_2 &= \frac{k_2}{\Delta x_2}, \text{ and} \\ T_{\infty 2} &= T_2. \end{aligned} \tag{4.3}$$

In other words, the sink temperature T_∞ that appears in the Robin boundary condition of Ω_1 is the temperature at an interior point of Ω_2, and the heat transfer coefficient h is the ratio between the conductivity value at this point in Ω_2 and the distance Δx_2. This particular way of computing the parameter values is taken from the implementation of the chtMultiRegionFoam solver of OpenFOAM.

In Section 5.1 it is demonstrated through a validation case that using Robin interface conditions with these specific parameter values results in a good stability and convergence behaviour.

As mentioned earlier, whether the coupling can be done with Robin boundary conditions depends on whether the solver supports this type of boundary condition and whether it can be modified by the adapter code. All three solvers used in this thesis support Robin boundary conditions. In both CalculiX and Code_Aster it is available as a convective heat transfer boundary condition, which allows to model the thermal interaction with a fluid when running a standalone solid simulation.

4.1.3 Steady-State Coupling

This section describes the approach used for coupling the steady-state solvers, which is done differently for the fluid and the solid solvers.

The steady-state OpenFOAM solvers approach the solution gradually, in pseudo time steps. In each pseudo time step, the residual of each field is reduced by a prescribed factor. The coupling boundary data is exchanged at the end of each pseudo time step of the fluid solver. In the case of the solid solvers, the full steady-state solution is computed during each coupling iteration. The computations for the solid are cheap enough to allow this, which is simpler than having to deal with intermediate solutions that are not yet converged, as in the fluid domain. The steady-state fluid flow equations are cheaper to solve than the transient ones, but they are still expensive and time-consuming, and therefore it is not practical to obtain the full steady-state solution at each coupling iteration.

In the current implementation of the steady-state coupling, only explicit coupling with Robin-Robin boundary conditions is supported. The reason for using explicit coupling is that implicit coupling did not seem to work well with the partially solved fluid fields (although future work may attempt to make implicit coupling work, so that convergence acceleration schemes can be used). Robin coupling boundary conditions are used, after the initial testing showed that they are stable even with explicit coupling. Convergence of the coupling can be monitored in terms of the convergence of the solution of the fluid flow fields.

The way the steady-state simulation is enabled, varies depending on the specific solver used. For example, in OpenFOAM, to do steady-state simulations, a different solver must be in-

voked, which means that a separate solver must be adapted. In CalculiX, to enable steady-state simulation, it is enough to add a keyword in the CalculiX input configuration file, and include minor changes in the adapter code. In Code_Aster, also minor changes in the adapter code is required, in order to invoke the steady-state solver.

4.2 Coupling of OpenFOAM

OpenFOAM[1] is a free, open-source CFD toolbox based on the finite volume method (FVM). It is developed primarily by OpenCFD Ltd. OpenFOAM is distributed with a standard set of solvers applicable to a specific class of problems (e.g. incompressible, compressible, combustion, multiphase, heat transfer, etc.), as well as utilities (e.g. pre-processing, post-processing, etc.) that are ready to be used.

For this thesis, solvers for heat transfer and buoyancy driven flows are used, namely:

- buoyantPimpleFoam – a transient solver for buoyant, turbulent flow of compressible fluids, which uses the PIMPLE algorithm. The PIMPLE algorithm is a combination of the PISO (Pressure Implicit with Splitting of Operator) algorithm [10] and the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm [11];
- buoyantSimpleFoam – a steady-state solver for buoyant, turbulent flow of compressible fluids, which uses the SIMPLE algorithm;
- buoyantBoussinesqPimpleFoam – a transient solver for buoyant, turbulent flow of incompressible fluids, which uses the Boussinesq approximation; and
- laplacianFoam – a Laplace equation solver.

OpenFOAM is not limited to the standard solvers, but can be used as a C++ library that allows users to compile their own applications (solvers and utilities), as well as libraries. These are called user applications and user libraries. For an overview of the principles behind the design and implementation of OpenFOAM as an object-oriented library for computational continuum mechanics problems, the reader is referred to [12]. The approach taken in this thesis for coupling OpenFOAM using preCICE is to:

1. Implement and compile an adapter *library* that can be used by several OpenFOAM solvers.
2. Copy existing solvers and modify them by adding calls to the adapter. Compile them as user *applications*.

The solvers that are of main interest for this thesis are buoyantPimpleFoam and buoyantSimpleFoam. Partially adapted and tested were buoyantBoussinesqPimpleFoam and laplacianFoam. Solvers that could be coupled in the future include rhoPimpleFoam and sonicFoam, for example. The idea is to create an adapter that is general enough, such that coupling new solvers requires minimal additions or no changes to the existing implementation.

OpenFOAM also has two CHT solvers: the transient solver chtMultiRegionFoam, based on the PIMPLE algorithm, and the steady-state solver chtMultiRegionSimpleFoam, based on the

[1] http://www.openfoam.com/

SIMPLE algorithm. These solvers are later used in some of the validation and demonstration cases in Chapter 5 and Chapter 6, respectively.

The structure of an OpenFOAM simulation case consists basically of three types of directories:

- The `constant` directory, which contains several files with the description of the mesh, as well as the physical properties of the fluid;
- The `system` directory, which contains configuration files for the solution procedure and the numerics; and
- Time directories, which contain files for each one flow fields. For example, the temperature boundary and internal fields at time $t = 0$ would be stored in `0/T`.

For the case to work correctly with the coupling, appropriate boundary conditions must be set to the temperature in the `T` file, in the directory of the starting time (usually `0/T`).

4.2.1 Description of the Adapter

The coupling adapter library developed for OpenFOAM consists of:

- The main `Adapter` class, which contains wrappers to the preCICE functions, takes care of advancing the coupling, triggering the reading and writing of data from one or more `Interfaces`, adjusting the time step size, and doing the checkpointing;
- The `Interface` class, which contains the data regarding an individual coupled surface, such as the number of data locations, the IDs of the boundary patches, the preCICE mesh name and ID, etc. An `Interface` may have multiple `CouplingDataReaders` and/or `CouplingDataWriters`; and
- `CouplingDataReader` and `CouplingDataWriter` classes, which implement the accesses to the boundary data. A `CouplingDataReader` *reads* the data from a buffer and sets it as a boundary condition, whereas a `CouplingDataWriter` extracts the values from the boundary and *writes* it into a buffer. The specific data that is accessed and the way it is manipulated is implemented by subclasses of `CouplingDataReader` and `CouplingDataWriter`. The adapter does not need to know what data is exchanged or how it is computed; for example, calling `writeCouplingData()` on the adapter will call `writeCouplingData()` on each interface, which in turn will call `write()` on each of its `CouplingDataWriters`.

During the initial setup of the coupled simulation, an `Adapter` must be instantiated, interfaces must be added to it, and for each interface, `CouplingDataReaders` and `CouplingDataWriters` have to be added. Solver-specific adapters can also be created by inheriting from the `Adapter` class.

Figure 4.2 shows a simplified class diagram, with the main classes and their members, as well as the relationships between them. This structuring of the adapter implementation should allow to easily adapt new solvers and add new types of coupling data or boundary conditions, allowing even for other types of multi-physics problems.

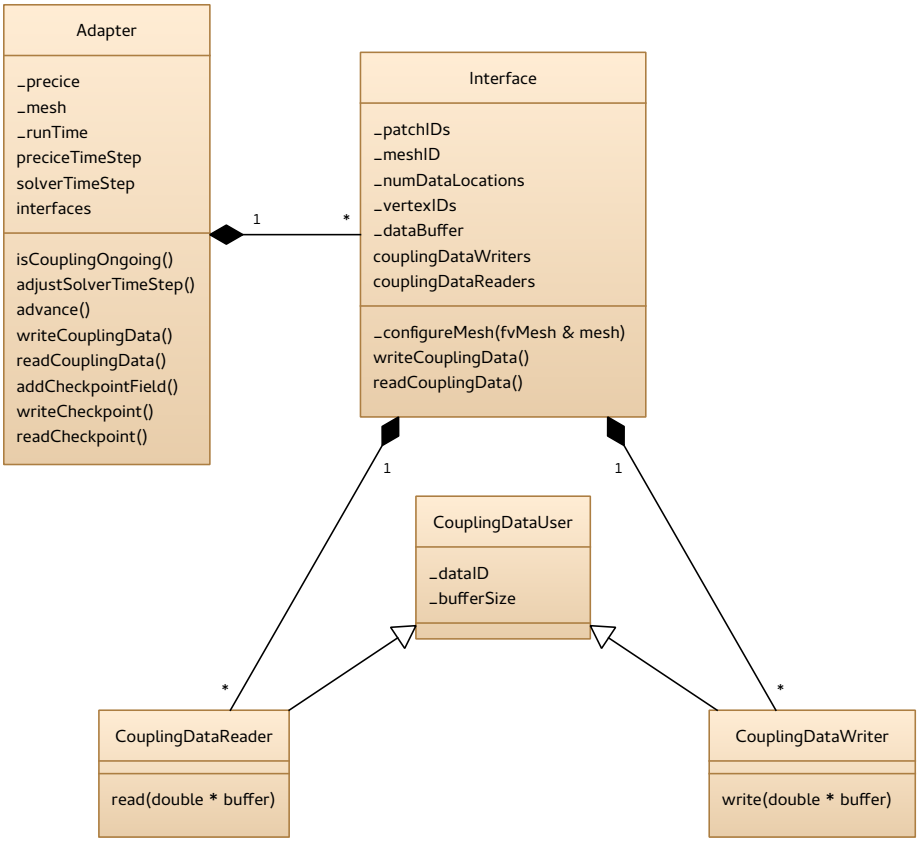


Figure 4.2: Simplified class diagram of the OpenFOAM adapter

4.2.2 Modifications to the Solver Code

The solver which is the main focus of this thesis, buoyantPimpleFoam, consists of a time stepping loop and an inner-loop which implements the PIMPLE algorithm, which solves the equations iteratively. Listing 4.1 shows the main parts of the loop that will be modified in the buoyantPimpleFoam solver.

```
while(runTime.run())
{
    ...
    #include "setDeltaT.H"
    runTime++;
    #include "rhoEqn.H"
    while (pimple.loop())
    {
        #include "UEqn.H"
        #include "EEqn.H"
        while (pimple.correct())
        {
            #include "pEqn.H"
        }
        ...
    }
}
```

Listing 4.1: Excerpt of the buoyantPimpleFoam solver

The changes that need to be performed in the solver loop are:

- The control of the time stepping is handed to preCICE by changing the loop condition to `adapter.isCouplingOngoing()`.
- After the solver time step is computed in `setDeltaT.H` (either set to a constant value or computed based on the Courant number [13]), and before the solver time is incremented by this value (`runTime++`), it needs to be adjusted, taking into account the maximum step size allowed by the coupling (i.e. if sub-cycling is used, the sub-steps must add up to the coupling time step so that the solvers can synchronize). Therefore, the method `adapter.adjustSolverTimeStep()` must be called between these two actions. The checkpoint must also be written here, by calling `adapter.writeCheckpoint()`.
- Before the equations are solved (i.e. before `rhoEqn.H`), the coupling data must be read with `adapter.readCouplingData()`.
- After the equations are solved, the updated coupling data must be written to preCICE with `adapter.writeCouplingData()`. The coupling is then advanced by calling `adapter.advance()`.
- After advancing, the coupling convergence is determined, and the checkpoint is read in the case it is necessary, with `adapter.readCheckpoint()`.

One important point to mention here, is that even though in many solvers with the double loop structure it is possible to avoid checkpointing, in the case of buoyantPimpleFoam it was found that the coupling did not converge well when checkpointing was not used. However, buoyantBoussinesqPimpleFoam did converge well without the checkpointing. This behaviour is at the moment not well understood, and therefore requires further investigation.

The checkpointing for the OpenFOAM solvers consists in storing the value of the current time, and making a copy of the relevant fields. At the beginning of the solver code, the fields that need to be checkpointed are registered with the adapter by calling `addCheckpointField(field)`.

Compared to the solid solver, where only the temperature T field is solved, the fluid solver is more complex, involving multiple fields (e.g. density, temperature, velocity, pressure), and the number increases if turbulence is modelled (e.g. k, ϵ). Therefore, it is desirable to avoid the checkpointing, which is expensive, even if this prevents using subcycling. Another motivation to avoid checkpointing if possible, is that if the old field values are not reloaded, the partial solutions are better "initial guesses" for the PIMPLE algorithm, which should translate into less effort (iterations) required by the individual matrix solvers, and therefore, a faster execution of the PIMPLE loop.

The reason why all the checkpointing related code is not moved into the adapter (e.g. the entire `if(adapter.isReadCheckpointRequired())` clause), is that there may be solver-specific actions that need to be included in the checkpointing. For example, in the case of buoyantPimpleFoam it was found that the method `correctBoundaryConditions()` of the turbulent thermal diffusivity `alphat` had to be called, otherwise, the convergence of the coupling would be affected.

4.2.3 Surface Mesh and Coupling Data

Data Locations

A mesh in OpenFOAM can consist of any type of polyhedra, although in most cases it consists of mostly hexahedral elements. All the coupling data from OpenFOAM is extracted from the face centers.

The values at the cell faces are extrapolated from the cell interior values. Different numerical schemes are available in OpenFOAM, and can be selected in runtime. Usually, by default, linear interpolation is used. When computing the surface normal gradient, schemes with non-orthogonal correction are typically used. The numerical schemes are documented in [14].

Computation of the Thermal Conductivity

The value of the thermal conductivity is required for the computation of the heat flux and the heat transfer coefficient. The computation of the conductivity varies depending on the solver:

- laplacianFoam: it solves the Laplace equation with only one material property, the thermal diffusivity α. In order to give a value to the conductivity, the solver must be mod-

```
while (adapter.isCouplingOngoing())
{
    ...
    #include "setDeltaT.H"

    adapter.adjustSolverTimeStep();
    if( adapter.isWriteCheckpointRequired() )
    {
        adapter.writeCheckpoint();
        adapter.fulfilledWriteCheckpoint();
    }

    runTime++;

    adapter.readCouplingData();

    #include "rhoEqn.H"
    while (pimple.loop())
    {
        #include "UEqn.H"
        #include "EEqn.H"
        while (pimple.correct())
        {
            #include "pEqn.H"
        }
        ...
    }
    ...

    adapter.writeCouplingData();
    adapter.advance();

    if( adapter.isReadCheckpointRequired() )
    {

        adapter.readCheckpoint();
        if( turbulenceUsed && adapter.isCheckpointingEnabled() )
        {
            turbulence->alphat()().correctBoundaryConditions();
        }
        adapter.fulfilledReadCheckpoint();

    }
    ...

}
```

Listing 4.2: Excerpt of the adapted buoyantPimpleFoam solver

ified to read the density ρ and the specific heat capacity c_p from the input files. The conductivity is then computed as $k = \alpha \rho c_p$.

- buoyantBoussinesqPimpleFoam: the effective conductivity k_{eff} is computed as

$$k_{eff} = \alpha_{eff} \rho c_p = (\alpha + \alpha_t) \rho c_p = \left(\frac{\nu}{Pr} + \frac{\nu_t}{Pr_t} \right) \rho c_p \tag{4.4}$$

  where $\alpha_t = \nu_t / Pr$ is the turbulent thermal diffusivity. The material properties ρ and c_p also need to be added to the solver.

- buoyantPimpleFoam/buoyantSimpleFoam: The effective conductivity is the same as in 4.4, although for this solver this value can be accessed directly by calling the method `kappaEff()` on the turbulence object (i.e. `turbulence->kappaEff()`).

Boundary conditions

Depending on the boundary conditions used for the coupling, the appropriate boundary condition must be assigned to the interface during the case setup[2] (e.g. in the `0/T` file).

- A **Dirichlet** boundary condition is of type `fixedValue`. The value of the temperature obtained from the coupling is applied directly. The class `TemperatureBoundaryCondition`, subclass of `CouplingDataReader`, takes care of setting the values.

- A **Neumann** boundary condition is of type `fixedGradient`. The value to be applied is the value of the heat flux sent by the other participant, divided by the conductivity. There are different subclasses of `CouplingDataReader` that encapsulate the details of computing the conductivity (which varies depending on the solver) and the gradient.

- A **Robin** boundary condition is of type `mixed`, which is a combination of `fixedValue` and `fixedGradient`, and has the form

$$T_{face} = f T_{ref} + (1 - f)(T_{center} + g_{ref} * \delta) \tag{4.5}$$

  where T_{face} is the patch value, T_{center} is the internal cell value, $0 \leq f \leq 1$ is a weight (called *value fraction*), and δ is the face center to cell center distance. T_{ref} denotes a reference value and g_{ref} denotes a reference gradient.

  To model a convective boundary condition using the `mixed` type boundary condition, appropriate values for the parameters T_{ref}, g_{ref} and f must be used. To do this, the derivative in (2.15) is linearized

$$k \frac{T_{face} - T_{center}}{\delta} + h(T_{face} - T_\infty) = 0, \tag{4.6}$$

  and an expression for T_{face} is obtained

$$T_{face} = \frac{h}{k/\delta + h} T_\infty + \frac{k/\delta}{k/\delta + h} T_{center}. \tag{4.7}$$

[2] `http://www.openfoam.com/documentation/user-guide/standard-boundaryconditions.php`

Comparing 4.5 and 4.7, the following values for the parameters of the `mixed` boundary condition are obtained:

$$T_{ref} = T_\infty, \tag{4.8}$$

$$f = \frac{h}{k/\delta + h} = \frac{h_{neighbor}}{h_{self} + h_{neighbor}}, \text{ and} \tag{4.9}$$

$$g_{ref} = 0. \tag{4.10}$$

Recall that h is a coupling variable, which means that it is sent by the partner participant, whereas the gradient is computed locally. For this reason, the variables have been subscripted with "self" and "neighbor". This implementation is based on OpenFOAM's implementation found in turbulentTemperatureCoupledBaffleMixedFvPatchScalarField.C[3], which is used by the chtMultiRegionFoam solver. This is also explained in [15].

Note that the closer f is to 1, the more the boundary condition resembles a Dirichlet boundary condition. This happens if the conductivity k of the other domain (neighbor) is very high, or if the distance δ at which the T_∞ is obtained is very small (meaning that it is basically the interface temperature).

The classes `SinkTemperatureBoundaryCondition` and `HeatTransferCoefficientBoundaryCondition` take care of updating these values in the boundary condition of type `mixed`.

Boundary values

- **Temperature**: Does not require special handling, can be extracted directly from the boundary field of the temperature variable.
- **Heat flux**: The heat flux is computed as the surface normal gradient of the temperature field multiplied by the conductivity. The surface normal gradient is computed with the built-in method `snGrad()`. The computation of the conductivity depends on the specific OpenFOAM solver used, as explained above.
- **Sink temperature** and **heat transfer coefficient**: As sink temperature, the cell center temperature is used. In order to access this value, the built-in method `patchInternalField()` is used to access the cell value associated to a boundary patch of type `fvPatchScalarField`. The heat transfer coefficient is the conductivity divided by a distance δ. The value of δ can be obtained through the method `deltaCoeffs()`[4], which gives the reciprocal of the cell-to-cell (or cell-to-face) distance, with orthogonality correction.

[3] `https://github.com/OpenFOAM/OpenFOAM-2.1.x/blob/master/src/turbulenceModels/compressible/turbulenceModel/derivedFvPatchFields/turbulentTemperatureCoupledBaffleMixed/turbulentTemperatureCoupledBaffleMixedFvPatchScalarField.C`

[4] `https://openfoamwiki.net/index.php/OpenFOAM_guide/SurfaceInterpolation;;deltaCoeffs`

4.2.4 Steady-State Simulations

To solve steady-state problems, the coupled buoyantSimpleFoam solver must be used.

4.2.5 Parallelization

OpenFOAM supports domain decomposition[5]. There is no overlap between the sub-domains and therefore no special handling is necessary. It is enough to specify the MPI rank and the MPI size at the initialization of preCICE in order to run OpenFOAM as a parallel participant.

4.3 Coupling of CalculiX

CalculiX[6] is a finite element package that can solve static, dynamic and thermal problems. The project is developed and maintained by Guido Dhont and Klaus Wittig.

The source code is mainly written in C and Fortran. Depending on the type of analysis (e.g. static, frequency, buckling, dynamic, electromagnetic, etc.), different solvers are called. The solvers are implemented as C functions that are called from the program's `main` function. The solver to be modified in order to do the thermal coupling is the `nonlingeo` function, which handles all the non-linear problems. Since the material properties might be temperature dependent, all heat transfer problems are treated by the non-linear solver.

The adapter was implemented in C, because the main solver code is in this language. Some additional Fortran functions were implemented to extract some of the coupling data.

To run a CalculiX simulation, an input (*.inp) file is required, which contains keywords with parameters and values, that describe the analysis to be performed. For heat transfer problems, the `*HEAT TRANSFER` keyword must be used. The parameter `DIRECT` can be added, in order to deactivate the automatic adjustment of the time step (i.e. `*HEAT TRANSFER, DIRECT`). Files that describe the mesh and the boundary conditions are included from the input file. Similar to OpenFOAM, in order for the coupling to work, the boundary condition for the interface must be properly defined in the case setup.

4.3.1 Description of the Adapter

Two C structures were added:

- `SimulationData`, which gathers the CalculiX variables that need to be accessed during the coupling. The idea is to avoid having to pass long lists of parameters in the function calls. A list of the CalculiX variables and their meaning is provided in [16] (page 518). Some other variables related to the coupling were also included in this structure.

[5] http://cfd.direct/openfoam/user-guide/running-applications-parallel/
[6] http://www.calculix.de/

- `PreciceInterface`, which contains the data belonging to a coupled surface (such as mesh names and IDs, vertex IDs, preCICE data IDs, etc.).

Additionally, three groups of C functions were added:

- `Precice_*` functions, which carry out coupling operations and are a wrapper to some preCICE functions;
- `PreciceInterface_*` functions, which handle the meshes and data of the individual coupled surfaces;
- CalculiX helper functions, which are lower level functions that manipulate CalculiX's data.

Compared to the OpenFOAM adapter, the `Precice_*` functions correspond to the `Adapter` class in the OpenFOAM adapter, and the `PreciceInterface_*` functions correspond to the `Interface` class. The most relevant functions of the CalculiX adapter are listed in Table 4.1.

`Precice_Setup`
`Precice_InitializeData`
`Precice_AdjustSolverTimeStep`
`Precice_IsCouplingOngoing`
`Precice_ReadCouplingData`
`Precice_WriteCouplingData`
`Precice_Advance`
`Precice_ReadIterationCheckpoint`
`Precice_WriteIterationCheckpoint`
`PreciceInterface_Create`
`PreciceInterface_ConfigureFaceCentersMesh`
`PreciceInterface_ConfigureNodesMesh`
`PreciceInterface_ConfigureTetraFaces`
`PreciceInterface_ConfigureHeatTransferData`

Table 4.1: Main functions of the CalculiX adapter

The CalculiX Fortran function `printoutface` which is used for writing surface values to a file during a simulation run, was copied and modified to extract coupling data:

- `getflux`: extracts the heat flux values at the specified surfaces
- `getkdeltatemp`: extracts the heat transfer coefficient and the sink temperature at the specified surfaces

4.3.2 Modifications to the Solver Code

The algorithm for the non-linear calculations is explained in CalculiX's user manual [16]. The structure consists of an *increment loop* (or time-stepping loop) and an *iteration loop*, where the solution of one time step is obtained iteratively.

Adapting CalculiX for CHT using preCICE consists in adding calls to the `Precice_*` functions inside the `nonlingeo.c` file:

- First, a `SimulationData` structure is created and populated with the CalculiX variables that need to be accessed during the coupling.
- The function `Precice_Setup()` must be called, which takes care of configuring and initializing preCICE, as well as the coupling interfaces.
- The increment loop condition is modified (`while((1.-theta>1.e-6)||(negpres==1))`), so that the time stepping is controlled by the adapter (`Precice_IsCouplingOngoing()`).
- At the beginning of the increment loop:
  - Read the coupling data with `Precice_ReadCouplingData`.
  - Write the iteration checkpoint if necessary (store the temperature field and the current time), with `Precice_WriteIterationCheckpoint`. The temperature field is stored in an array, therefore, a copy of it is made with the C function `memcpy`. The simulation time is also stored by copying the value of the `theta` variable, which is the CalculiX variable for the current time, normalized with respect to the total time.
- At the end of the increment loop:
  - Write the coupling data with `Precice_WriteCouplingData`.
  - Read the iteration checkpoint if necessary (reload the temperature field and the stored time), with `Precice_ReadIterationCheckpoint`. The reading of the checkpoint is exactly the opposite procedure.

4.3.3 Surface Mesh and Coupling Data

The implementation of this thesis supports first- and second-order tetrahedral elements. Surface elements therefore consist of triangles. The temperature data is located at the element nodes, while the rest of the coupling data (heat flux, sink temperature and heat transfer coefficient) is located at the face centers. This means that when reading the temperature from the CalculiX participant, nearest-projection mapping method can be used. The rest of the coupling data can only be treated as a cloud of points, as they do not have connectivity information.

When Dirichlet-Neumann coupling is used (i.e. when temperature data is involved), the CalculiX participant has to define two surface meshes for preCICE: one for the nodes and one for the face centers. When using Robin-Robin coupling, only the face centers need to be defined.

Boundary conditions

There are two CalculiX arrays that contain the parameter values of the boundary conditions: `xboun`, which contains temperature values for Dirichlet boundary conditions, and `xload`, which contains heat flux, sink temperature and heat transfer coefficient values for Neumann and Robin boundary conditions. The indexing of these arrays must be handled carefully, in order to modify the value of the intended thermal variable, corresponding to the intended node or face. For this purpose, the helper functions `getXbounIndices` and `getXloadIndices` were implemented as part of the adapter.

- **Dirichlet**: The temperature boundary condition must be defined in the input file, using the keyword `*BOUNDARY`[7]. Temperature values are applied at the nodes. The temperature values are set with the helper function `setNodeTemperatures`.
- **Neumann**: A distributed heat flux boundary condition `*DFLUX`[8] must be used. The values for the distributed heat flux are set in the `xload` array with the helper function `setFaceFluxes`.
- **Robin**: a convective boundary condition `*FILM`[9] is used. The values are set in the `xload` array with `setFaceHeatTransferCoefficients` and `setFaceSinkTemperatures`.

Boundary values

- **Temperature**: the temperature data is directly obtained from the solution vector at the end of the inner solver iterations. The helper function `getNodeTemperatures` was implemented to extract the temperature values at the interfaces.

The CalculiX function `printoutface` was copied and modified to extract the heat flux, the sink temperature and the heat transfer coefficient. This function computes the variables at the surface Gauss points. In the modified copies of the function (`getflux` and `getkdeltatemp`), if first-order elements are used, the values are extracted from the only Gauss point that the face element has. If second-order elements are used, the average value between the three Gauss points is used. In both cases, only the face center location is exposed to preCICE.

- **Heat flux**: The function `printoutface` prints out the heat flux values at the Gauss points, as well as other variables if other type of analysis is selected. This was modified to extract only the heat flux at the face elements of the specified surfaces (i.e. the interfaces) and store the values in an array.
- **Sink temperature** and **heat transfer coefficient**: The same data that is used for calculating the heat flux, is used for calculating the sink temperature and the heat transfer coefficient. Therefore, the printoutface.f file was also used as a base to extract the variables for the Robin coupling:

$$h = \frac{k}{\delta}, \tag{4.11}$$

[7] `http://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node163.html`
[8] `http://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node188.html`
[9] `http://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node203.html`

$$T_{interior} = T_{face} + \frac{q\delta}{k}. \tag{4.12}$$

4.3.4 Steady-State Simulations

In order to do a coupled steady-state simulation, it is enough to add the the option `STEADY STATE` to the `HEAT TRANSFER` keyword card in the case input (*.inp) file, as shown in Listing 4.3. In the second line, the value of the time step size and the final time are to be specified. These values are not important for the coupled steady-state simulations. In the adapter function `Precice_AdjustSolverTimeStep`, at every coupling iteration the current time is set to 0 and the normalized step size ($\Delta t / t_{END}$) is set to 1. This way, it forces the solver to obtain the full steady-state solution in one step.

```
*HEAT TRANSFER, DIRECT, STEADY STATE
1, 1
```

Listing 4.3: Configuration for the steady-state heat transfer solver in CalculiX

4.3.5 Parallelization

At the moment of writing this thesis, CalculiX only supports multithreading for solving the system of equations. Therefore, from the point of view of preCICE, the CalculiX participant is just a serial participant.

4.4 Coupling of Code_Aster

Code_Aster[10] is a free and open-source finite element analysis package developed originally by Électricité de France (EDF).

Code_Aster's source code is mostly written in Fortran and Python. There are two ways one could couple Code_Aster. The first one is to directly modify the source code by introducing the calls to preCICE. The second option is to couple it at a higher level, adding the coupling calls to the command file that controls the flow of the simulation. This file contains Code_Aster commands[11] for creating the mesh and the model, for defining boundary and initial conditions, and calling the relevant solver. This command file can be extended with Python scripting to add more advanced functionality.

Listing 4.4 shows the typical structure of a Code_Aster command file. First, the case is created by defining the mesh, the model, the materials, and the boundary and initial conditions. Then a list of time steps is created, and is provided together with the setup to the solver.

[10] http://code-aster.org

[11] The documentation of all the commands can be found at `http://code-aster.org/doc/v12/en/index.php?man=commande`. Beware that the English version is based on machine translation. The original documentation is in French.

```
# Read the mesh
MESH = LIRE_MAILLAGE(...)

# Create the model
MODEL = AFFE_MODELE(..., MAILLAGE=MESH, ...)

# Define a material
MAT = DEFI_MATERIAU(...)

# Assign the material to the mesh
MATS = AFFE_MATERIAU(..., MAILLAGE=MESH,...)

# Create boundary conditions
BC = AFFE_CHAR_THER(..., MODELE=MODEL, ...)

# Create initial condition
INIT_T = CREA_CHAMP(...)

# Define a list of steps to solve
STEP = DEFI_LIST_REEL(...)

# Call the solver
TEMP = THER_LINEAIRE(..., MODELE=MODEL, ...)
```

Listing 4.4: Sample structure of a Code_Aster command file

For this thesis, it was decided to couple Code_Aster through the command file, since it is the least invasive way, as it does not require modifying the source code at all. Therefore, the coupling of Code_Aster consisted in the implementation of a command file which contains coupling operations and calls to the thermal solver. The linear thermal solver in Code_Aster is `THER_LINEAIRE` and the non-linear thermal solver is `THER_NON_LINE`.

The Code_Aster adapter was implemented after testing the different boundary conditions with the CalculiX adapter, and it was decided that only the Robin coupling boundary condition would be implemented for Code_Aster, since it performed better than the other ones.

4.4.1 Description of the Adapter

The approach taken in this thesis is to split the command file into two command files:

- The **case definition** command file, which is in charge of the creation of the mesh, model, materials, and the definition of the initial and boundary conditions;
- The **adapter** command file, which wraps the solver call in a loop and triggers the coupling operations. This file is the main command file. The case definition command file is read by the adapter through the `INCLUDE` command, invoked at the beginning.

Besides the adapter command file, a Python module was developed, which is in charge of carrying out lower level accesses to the solver's data for the coupling. Therefore, the Code_Aster adapter consists of two files:

- `adapter.comm`: contains the main solver loop with the commands to run the simulation, as well as the coupling operations;
- `adapter.py`: contains two classes:
  - `Adapter`, which handles the coupling;
  - `Interface`, which handles the coupled surface meshes and their data.

Listing 4.5 shows the general structure of the adapted Code_Aster. The solution process takes place in the call to `THER_LINEAIRE` for linear problems or `THER_NON_LINE` for non-linear problems. The procedure for steady-state problems is slightly different, but for simplicity, the checking of whether it is a linear or non-linear, steady-state or transient problem, has been omitted from the code snippet. Boundary conditions are updated before the call to the solver by calling `adapter.readCouplingData()`. Similarly, updated values are written to preCICE after solving, by calling `adapter.writeCouplingData()`. The time step size of the solver is always equal to the coupling time step size (returned by the `initialize` or `advance` methods of preCICE), which means that sub-cycling is not supported. This is, however, not a concern, because it is typically the fluid solver that needs sub-cycling. The `adapter.readCheckpoint()` and `adapter.writeCheckpoint()` functions do not really perform any checkpointing, but directly notify preCICE that the checkpoint reading/writing has been fulfilled. The solver advances in time only if the coupling converges.

```
...

# Include the case definition .comm file
INCLUDE(UNITE=91)
...

# Reset time and set initial condition
k = 0
time = 0.0
ICOND = {'CHAM_NO': INIT_T}

while precice.isCouplingOngoing():

    adapter.writeCheckpoint()
    adapter.readCouplingData()
    ...

    # Call the linear thermal solver
    TEMP = THER_LINEAIRE(MODELE=MODEL,
        CHAM_MATER=MATS,
        EXCIT=LOADS,
        ETAT_INIT=ICOND,
        INCREMENT=_F(LIST_INST=STEP),
        PARM_THETA=1.0
    )
    T = CREA_CHAMP(RESULTAT=TEMP,
        NOM_CHAM='TEMP',
        TYPE_CHAM='NOEU_TEMP_R',
        OPERATION='EXTR',
        NUME_ORDRE=1
    )

    adapter.writeCouplingData(T)
    dt = adapter.advance()
    adapter.readCheckpoint()

    if adapter.isCouplingTimestepComplete():

        # Output if necessary ...

        # Set current solution as initial condition of next time step
        ...
        TEMP_CPY = COPIER(CONCEPT=TEMP)
        ICOND = {'EVOL_THER': TEMP_CPY}

        # Increment time
        time = time + dt
```

Listing 4.5: Code_Aster adapter command file

4.4.2 Surface Mesh and Coupling Data

Thermal boundary conditions in Code_Aster are assigned through the AFFE_CHAR_THER[12] command. A convective boundary condition was used for the Robin-Robin coupling. In Code_Aster, this corresponds to using a boundary of type ECHANGE.

The convective boundary condition is applied to the element face, therefore, the face center is used as the data location. However, for extracting the sink temperature and the heat transfer coefficient, it was more straightforward to do it on the element nodes. Therefore, for the coupling of Code_Aster, two meshes are used: face centers for *read*-data and nodes for *write*-data.

To compute the sink temperature to be sent to the coupling partner, temperatures at the interior of the solid are sampled. To obtain the points inside the solid, the nodes of the surface mesh are taken as starting points. The position of each node is then displaced by a prescribed distance δ in the direction opposite to the surface normal. During the coupling, the value of the temperature at the interior point is obtained by interpolation, using the shape functions of the element that contains the point. This interpolation is performed with the Code_Aster operator PROJ_CHAMP[13], using the method COLLOCATION.

The configuration of the interfaces, the extraction and update of boundary values are performed in the `adapter.py` file. It must be noted that special methods[14] have to be used in order to access the data of the Code_Aster objects. This is because Code_Aster is mostly programmed in FORTRAN, and only the names of the objects are available at higher levels (i.e. in the command file). In order to access the data itself, it must be retrieved from Code_Aster's memory manager "JEVEUX". This makes the implementation of some operations not so straightforward, because the data cannot be read or written directly to the variables.

4.4.3 Steady-State Simulations

To run a steady-state simulation, it is enough to call the solver without specifying an initial condition (e.g. removing the parameter ETAT_INIT) from the solver call, as shown in Listing 4.6.

```
TEMP = THER_LINEAIRE(
    MODELE=MODEL,
    CHAM_MATER=MATS,
    EXCIT=LOADS
)
```

Listing 4.6: Invoking the steady-state solver of Code_Aster

[12] http://www.code-aster.org/doc/v11/en/man_u/u4/u4.44.02.pdf
[13] http://www.code-aster.org/doc/v11/en/man_u/u4/u4.72.05.pdf
[14] http://www.code-aster.org/V2/doc/v12/en/man_u/u1/u1.03.02.pdf

This is taken care of by the adapter, as it reads from the configuration whether it is a steady-state or a transient analysis.

4.4.4 Parallelization

In Code_Aster, it is possible to use both shared memory (OpenMP) and distributed memory (MPI) parallelization. This can be chosen at runtime, through the configuration of the solver[15]. Additionally, the keyword `PARTITION` in the command `AFFE_MODELE`[16] allows to define how the elements will be distributed in the parallel phases of Code_Aster.

In the adapter implemented in this thesis, the Code_Aster participant only supports shared memory parallelization. From preCICE's point of view it would be just a serial participant. Due to time limitations, it was not possible to find a straightforward way to do the domain decomposition and have each process access only its own part of the mesh. This is the first step required to support parallelism in the coupling adapter. This may be addressed in future work.

[15] `http://www.code-aster.org/doc/v12/en/man_u/u4/u4.50.01.pdf`
[16] `http://www.code-aster.org/doc/v12/en/man_u/u4/u4.41.01.pdf`

Chapter 5

Validation Cases

The implementation of the coupling adapters is validated with two validation cases. The first case is for forced convection, where the results are compared against what is reported in the literature. The second case is for natural convection, and the results are compared against a reference solution obtained with the OpenFOAM CHT solver, chtMultiRegionFoam. The results as well as the difficulties are discussed.

5.1 Forced Convection: Flow Over a Heated Flat Plate

The numerical validation case described in this section consists of an incompressible laminar flow over a flat plate of finite thickness, heated from below with a uniform temperature. The work from [17] is used as reference, both for the setup and the results comparison. This validation case serves to test the different adapters, the coupling boundary conditions, the convergence acceleration schemes and the data mapping schemes.

5.1.1 Case Setup

The computational domain is described in Figure 5.1. A colder fluid flows over a flat plate of length L, which is heated with a uniform temperature T_s from the bottom. The case is run for different combinations of the Reynolds number Re, the Prandtl number Pr and the conductivity ratio $k = k_s/k_f$. The thickness of the plate also plays a role in the heat transfer, as demonstrated in [17], but for the purposes of this thesis it has been set to a fixed value of $b = L/4$.

A time step size $\Delta t = 0.01$ was used for both the fluid and the solid participants, and also as the coupling time step size.

Mesh

The mesh used for this simulation is shown in Figure 5.2. Although the geometries coincide at the interface, the nodes are located at different positions (non-conforming). For the fluid

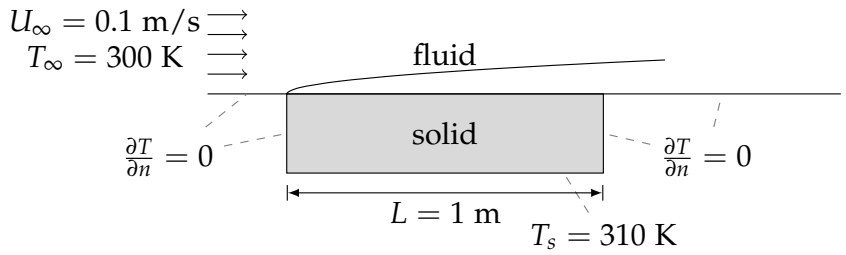


Figure 5.1: Geometry and boundary conditions of the flat plate case

domain, mesh grading was used in order to have a finer mesh near to the leading edge of the plate and at the fluid-solid interface, in order to properly capture the boundary layer. The fluid mesh consists of hexahedral elements and the solid mesh consists of first-order tetrahedral elements. Nearest-neighbor mapping is used at the interface.

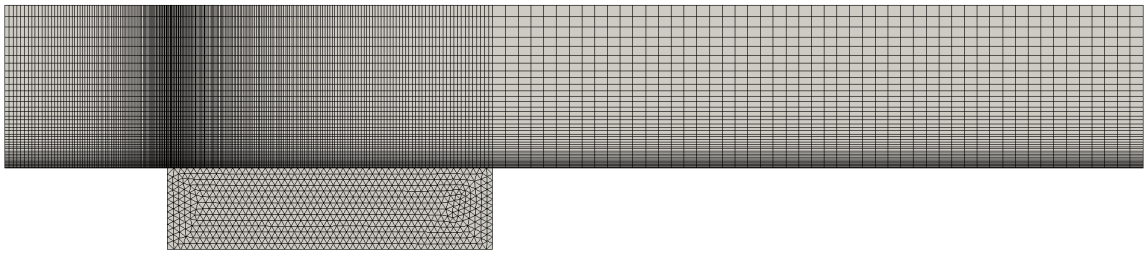


Figure 5.2: Meshing of the fluid and solid domains for the flat plate case

Boundary Conditions

The bottom of the plate is set to a uniform temperature of $T_s = 310$ K and the fluid inlet is set to a uniform temperature of $T_\infty = 300$ K. Apart from these two boundaries and the conjugate boundary, the thermal boundary condition for the rest of the faces is *adiabatic*.

The inlet velocity is set to a uniform value of $U_\infty = 0.1$. The bottom of the fluid domain is set to *slip* before the leading edge of the plate, and *no-slip* after it.

In order to simulate a two-dimensional setup, using the three-dimensional meshes, *symmetry* boundary conditions are used for the front and back faces.

Material and Flow Properties

A total of twelve cases were simulated, with different values for the three non-dimensional parameters *Re*, *Pr* and *k*. These are shown in Table 5.1.

Given that the solvers work with dimensional formulations, Table 5.2 shows the actual variables and the values that were used for the flow and material properties.

Case	Re	Pr	k
1	500	0.01	1
2	500	0.01	5
3	500	0.01	20
4	10000	0.01	1
5	10000	0.01	5
6	10000	0.01	20
7	500	100	1
8	500	100	5
9	500	100	20
10	10000	100	1
11	10000	100	5
12	10000	100	20

Table 5.1: Values for Re, Pr and k for the flat plate case

Parameter	**Symbol**	**Value**
Inlet velocity	U_∞	0.1
Plate length	L	1
Solid thermal conductivity	k_s	100
Solid specific heat capacity	c_{ps}	100
Solid density	ρ_s	1
Fluid thermal conductivity	k_f	$= k_s/k$
Fluid density	ρ_f	1
Dynamic viscosity	μ	$= \rho_f U_\infty L/Re$
Fluid specificy heat capacity	c_{pf}	$= k_f Pr/\mu$

Table 5.2: Flow and material properties for the flat plate case

5.1.2 Results

Temperature Profiles

The interface temperature profiles for the different combinations of Re, Pr and k are shown in Figures 5.3a to 5.3d. The results are shown for OpenFOAM - CalculiX *steady-state* coupling with Robin boundary conditions. The non-dimensional temperature θ is plotted against the distance x from the leading edge of the plate. The non-dimensional temperature is defined as

$$\theta = \frac{T - T_\infty}{T_s - T_\infty}. \tag{5.1}$$

A good agreement was found with the results reported by Vynnycky et al. [17]. The analytical solutions derived by [17] using boundary layer theory have also been plotted, as a reference. Note that cases 10-12 required a finer fluid mesh near to the interface, compared to the base mesh shown in Figure 5.2, in order to be able to reproduce the interface temperature from the reference.

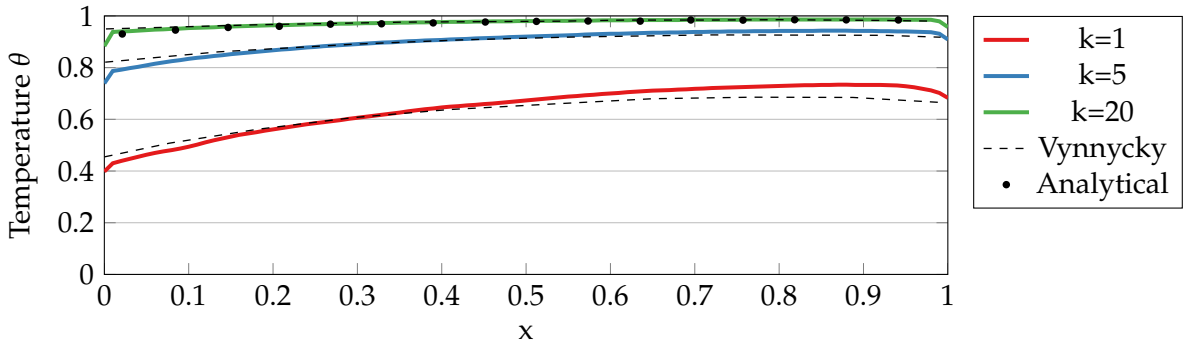
Coupling configurations that worked well, without requiring further changes to the base setup described in the previous section, were OpenFOAM Dirichlet - CalculiX Neumann (transient[1]) and OpenFOAM Robin - CalculiX Robin (transient and steady-state). Some difficulties were found in the validation of the temperature profiles for the other coupling configurations. The common behaviour among the cases that did not work well was that the temperature profile suffered from oscillations near to the leading edge of the plate, where larger gradients and a stronger mesh grading are present. The oscillations increased as more coupling iterations were performed, and the solvers finally diverged. In order to control or eliminate the oscillations, one or more of these changes proved to be effective for this validation case:

- Using a finer solid mesh
- Using second-order elements for the solid mesh
- Using thin plate spline radial basis functions (RBF-TPS) for the data mapping instead of nearest-neighbor (NN)

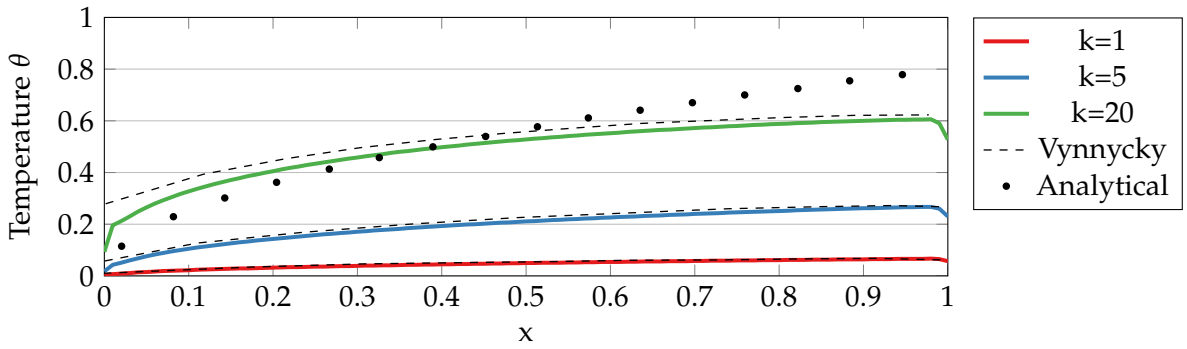
In the case of OpenFOAM Neumann - CalculiX Dirichlet (transient[2]) coupling, it was necessary to use second-order elements for the solid mesh, and RBF-TPS for the mapping, in order to eliminate the oscillations and prevent the solver from diverging. The effect of using second-order elements and RBF-TPS is shown in Figure 5.4, where the non-dimensional temperature is plotted at $t = 0.05$. As can be seen, by only changing the data mapping to RBF-TPS, the oscillations were significantly reduced, but it was still not enough to prevent the solver from diverging.

A similar situation was found when coupling with Code_Aster as the solid solver. The interface temperature was very sensitive to the mapping of the data between the surface meshes.

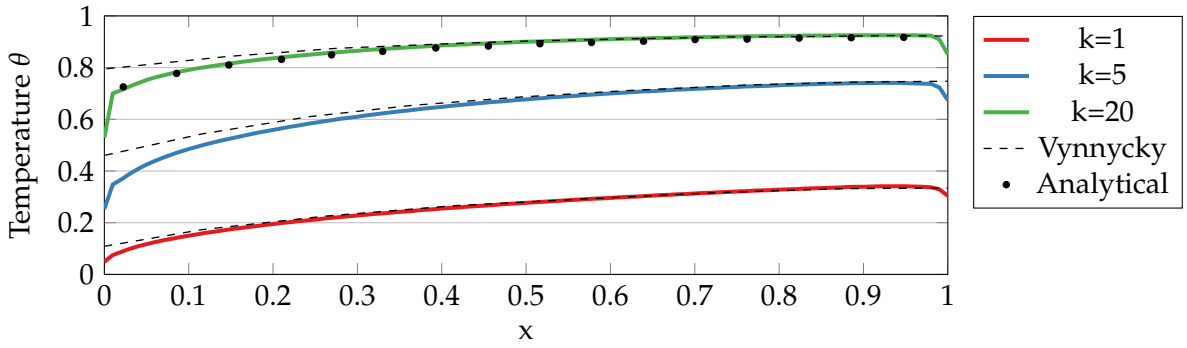
[1] Dirichlet-Neumann coupling has only been implemented for the transient solvers.
[2] See footnote 1



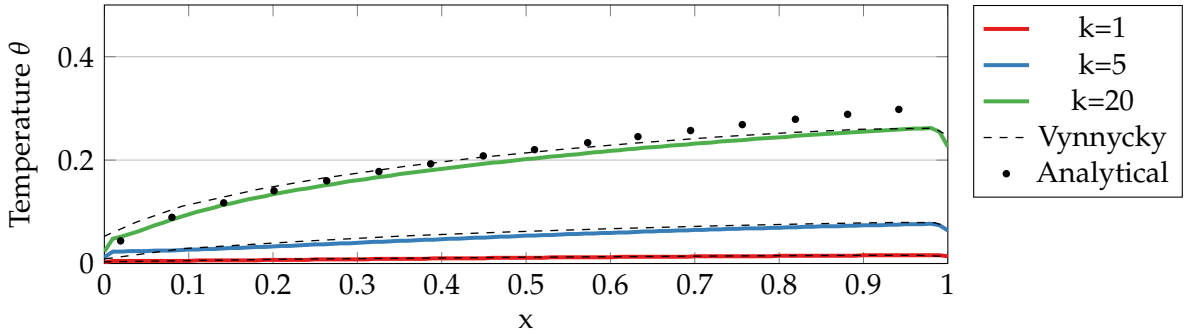
(a) Interface temperature for Re=500, Pr=0.01 (Cases 1, 2 and 3)



(b) Interface temperature for Re=500, Pr=100 (Cases 4, 5 and 6)



(c) Interface temperature for Re=10000, Pr=0.01 (Cases 7, 8 and 9)



(d) Interface temperature for Re=10000, Pr=100 (Cases 10, 11 and 12)

Figure 5.3: Interface temperature for different *Re*, *Pr* and *k*, for the flat plate case

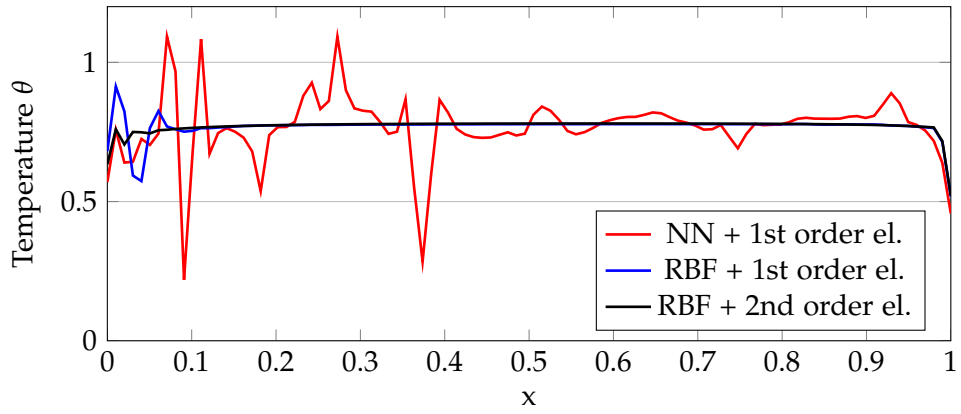


Figure 5.4: Interface temperature obtained with OpenFOAM Neumann - CalculiX Dirichlet coupling, using different mapping schemes: nearest-neighbor (NN) and thin plate spline radial basis functions (RBF-TPS), and elements of different order

Using the techniques mentioned above solved the problem. In Figure 5.5 the difference between using NN and RBF-TPS as mapping method is shown. The interface temperature was measured after 50 steady-state coupling iterations. With NN mapping the interface temperature profile was not smooth, and this eventually led to the divergence of the solvers. Using RBF-TPS for the mapping resulted in a smooth temperature profile.

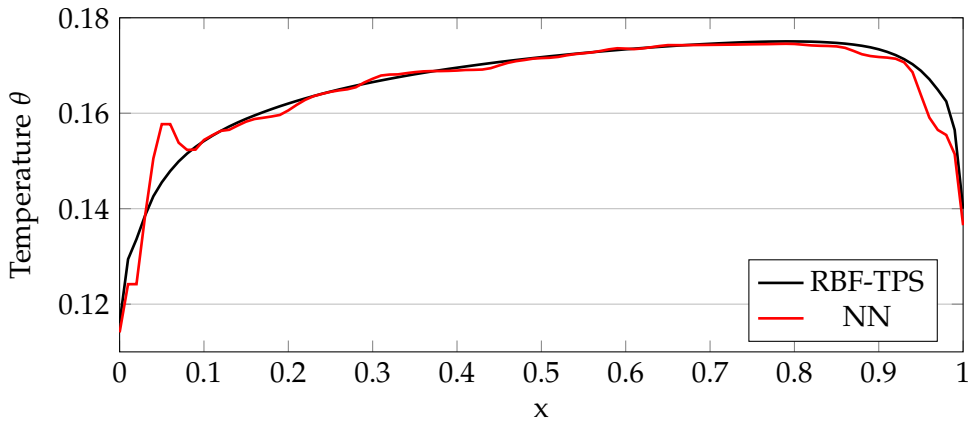


Figure 5.5: Interface temperature obtained with Code_Aster as solid solver, using different mapping schemes: nearest-neighbor (NN) and thin plate spline radial basis functions (RBF-TPS)

The vertical cross-sectional temperature profiles at $x = 0.5$ from the leading edge of the plate are shown in Figure 5.6 for some selected cases. The influence of the different parameters can be seen. The continuity of the temperature is evident, as there are no jumps at the interface. The continuity of the heat flux is visible in the plot for the case where $k = 1$, since in this case the gradients are equal on both sides (recall (2.18)).

Due to the large number of possibilities for the coupling configuration and the number of cases, it is not practical to show all the results in this document. Therefore, a summary is

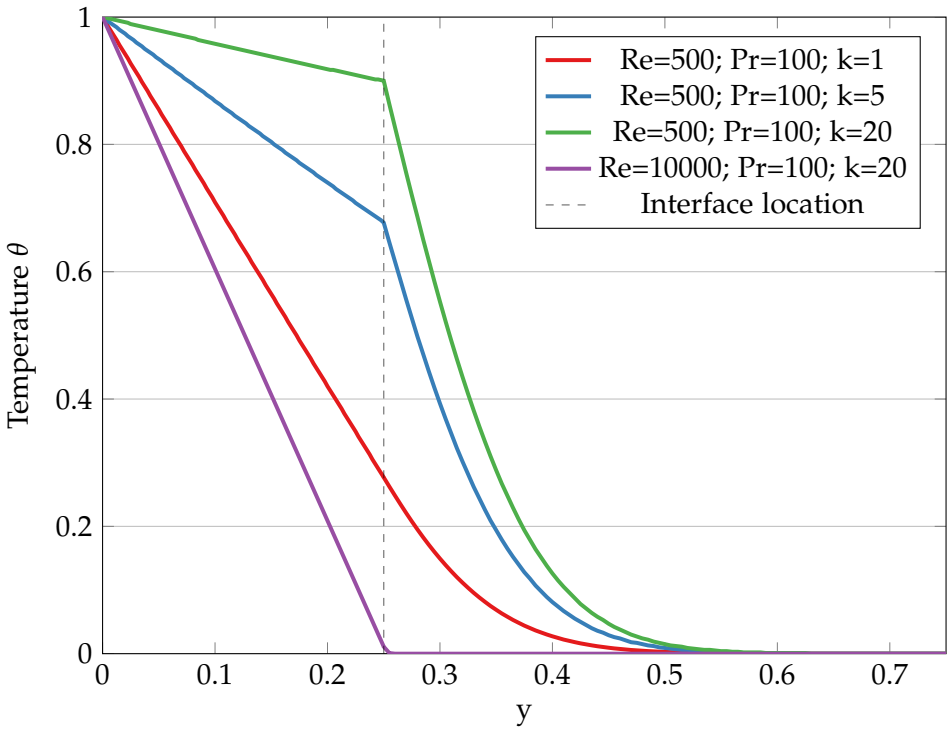


Figure 5.6: Cross sectional temperature profile for the flat plate validation case, at $x = 0.5$ from the leading edge, for different Re, Pr and k

provided in Table 5.3 and Table 5.4 for transient and steady-state coupling, respectively, regarding some of the findings and problems that were encountered in the course of validating the coupling using the heated flat plate case.

Solvers	**Coupling**	**Comments**
OpenFOAM - CalculiX	Fluid Dirichlet - Solid Neumann	The temperature profiles were successfully validated using implicit coupling, and nearest-neighbor (for the heat flux) and nearest-projection data mapping (for the temperature).
	Fluid Neumann - Solid Dirichlet	Some cases were unstable. Using second-order elements for the solid and RBF-TPS mapping worked.
	Robin - Robin	All cases were successfully validated. Worked well with nearest-neighbor mapping. Very stable. All 12 cases could be run also with explicit coupling.
OpenFOAM - Code_Aster	Robin - Robin	Some cases were unstable. Better results were obtained by using a finer solid mesh.

Table 5.3: Comments regarding the results of the transient coupling

Solvers	Coupling	Comments
OpenFOAM - CalculiX	Robin - Robin	All cases worked without problems.
OpenFOAM - Code_Aster	Robin - Robin	Oscillations and instabilities appeared in some cases when using nearest-neighbor mapping. By using RBF-TPS mapping in all cases, the temperature profiles were successfully validated.

Table 5.4: Comments regarding the results of the steady-state coupling

Coupling Iterations

Transient simulations were also run, and results were obtained for the first 100 time steps, for OpenFOAM - CalculiX implicit coupling. The influence of the coupling boundary condition and the convergence acceleration scheme on the number of iterations is shown in Figure 5.7. As can be seen, the convergence rate also varies with the material and flow properties.

Among the coupling boundary conditions that have been implemented and tested, the most robust one was the Robin-Robin coupling. It required the least average number of iterations and for all cases it was stable even with explicit coupling. In the literature it is very common to find Dirichlet-Neumann coupling boundary conditions for CHT problems, but it is known that they often present numerical instabilities [4].

The results also show that using IQN-ILS[3] as convergence acceleration scheme reduced considerably the number of coupling iterations required, compared to using Aitken underrelaxation.

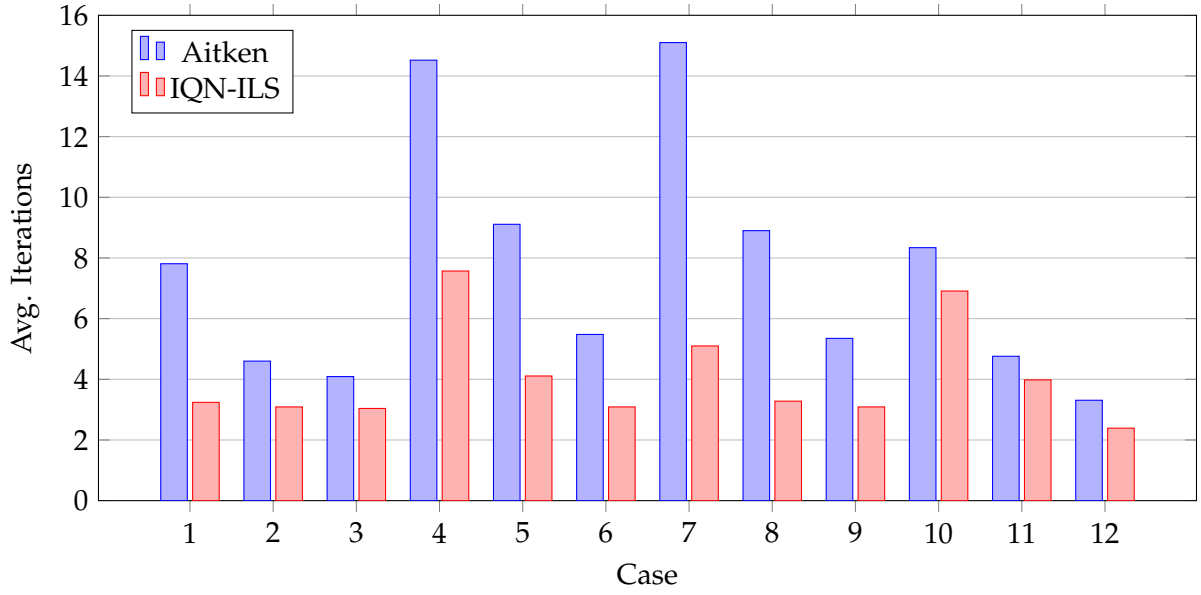
5.1.3 Final Remarks

Even though the geometry of this validation case is very simple, the strong grading of the fluid mesh and the larger gradients near to the leading edge of the plate, as well as the difference in size between the fluid and the solid elements, can pose difficulties to the convergence of the coupling.

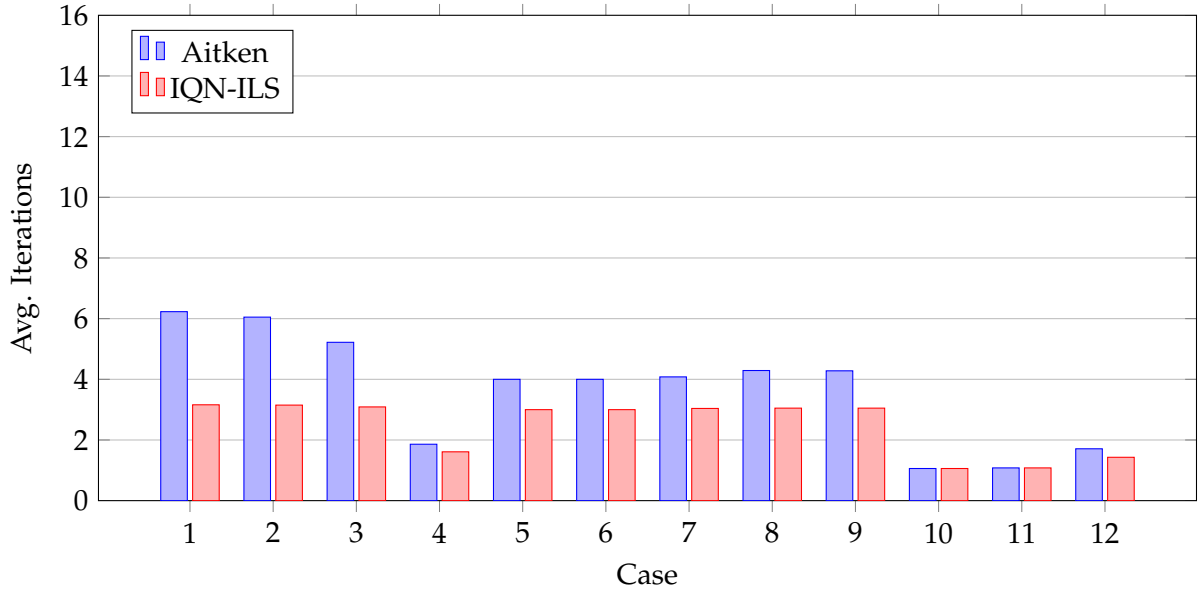
Overall, the setting that worked best was Robin-Robin coupling, with IQN-ILS for convergence acceleration. As shown in the graphs from Figure 5.7, the number of iterations are significantly lower than with Aitken or Dirichlet-Neumann coupling. It was also found that the convergence was good without initial under-relaxation (setting the initial under-relaxation to 1 in the configuration of preCICE), when using Robin-Robin coupling with IQN-ILS.

From the experiments, it was found that the OpenFOAM - CalculiX coupling worked better than OpenFOAM - Code_Aster, which was more sensitive to the mesh and the data-mapping scheme used. This does not mean that one solid solver is superior to the other one; instead, it should be taken as an indication that there is room for improvement for the Code_Aster adapter, specially with regards to how and where the coupling data is computed. It is not

[3]Configured with max-used-iterations=100, timesteps-reused=10 and initial-relaxation=1



(a) Fluid Dirichlet - Solid Neumann coupling



(b) Robin-Robin coupling

Figure 5.7: Average number of coupling iterations for the first 100 time steps, for the flat plate case

clear to what extent the instabilities are due to having different locations for the read- and write-data, therefore, further investigation into this aspect is still necessary.

For this CHT solver to be usable in the SimScale platform in the future, it is important to identify robust setups, so that the users of the platform can focus on the physics of the problem, and not on the numerics of the coupling. This case has been useful to identify what works well, what does not work well and how to improve it.

Given the observations and findings from this validation case, the subsequent validation and demonstration cases are carried out with CalculiX as the solid solver, with Robin-Robin coupling boundary conditions, unless otherwise stated.

5.2 Natural Convection: Cavity with Heat-Conducting Walls

The previous validation case consisted of forced convection over a flat plate, where the gravity was not taken into account. The case that will be presented in this section takes into account the gravity, and therefore, buoyancy forces. The validation is done by comparing the results with OpenFOAM's CHT solver, chtMultiRegionFoam. Natural convection inside a conductive enclosure is simulated. Similar setups have been studied experimentally by [18], and numerically by [19], [20] and [21]. As explained in [19], natural convection in enclosures has many thermal engineering applications, including for example the cooling of electronic devices and building insulation.

The main goal of this case is to validate the transient solution, therefore, the analysis will be focused on comparing the solution fields at different times.

5.2.1 Case Setup

The geometry and the boundary conditions are shown in Figure 5.8.

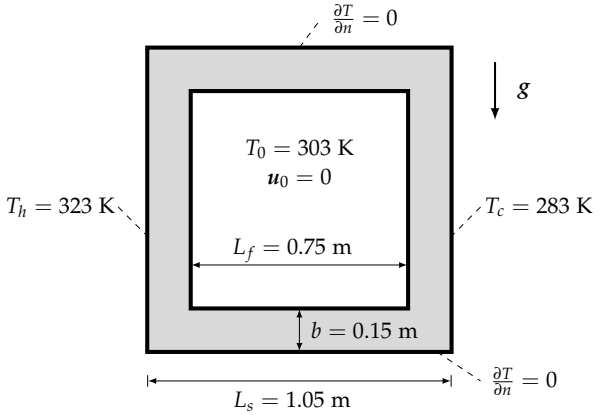
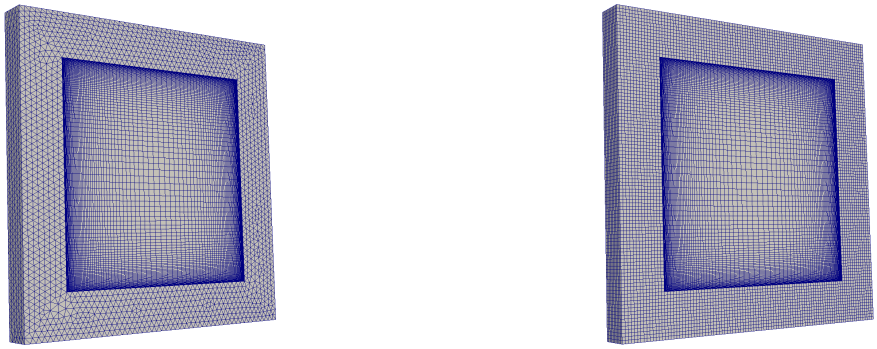


Figure 5.8: Geometry and boundary conditions of the cavity case

The left wall of the enclosure has a fixed temperature of $T_h = 323$ K, and the right wall has a fixed temperature of $T_c = 283$ K. The initial temperature is set to $T_0 = 303$ K for both the fluid

and the solid domains. The fluid is initially at rest. The top, bottom, front and back faces of the domain are adiabatic. A $k - \epsilon$ turbulence model was used for this simulation.

The case setup is two dimensional, although the meshes are three dimensional. The fluid mesh is purely hexahedral, with grading near to the walls. The solid mesh for the coupling via preCICE consists of first-order tetrahedral elements, whereas the solid mesh for the reference solution with chtMultiRegionFoam consists of hexahedral elements. The meshes are shown in Figure 5.9.



(a) Mesh used with the coupling via preCICE (b) Mesh used with chtMultiRegionFoam

Figure 5.9: Meshes for the cavity case

The coupling step size is set to 1. Sub-cycling is used for the fluid solver, where the step size is automatically adjusted by OpenFOAM based on the CFL condition [13] to ensure stability. Given that the fluid starts out at rest and gradually accelerates, using an adjustable time step allows to use larger time step sizes when the velocities are low, and thus save some computation time. The rest of the setup for the coupling is described in Table 5.5.

It must be noted that OpenFOAM's CHT solver only supports explicit coupling, where the number of coupling iterations can be specified. In this case, 10 coupling iterations were used per time step. The configuration used for the coupling with preCICE is shown in Table 5.5.

Coupling parameters	
Solvers	OpenFOAM and CalculiX
Coupling boundary conditions	Robin-Robin
Coupling scheme	Serial-implicit
Coupling convergence acceleration	IQN-ILS (max-used-iterations=100, timesteps-reused=10)
Data mapping	Nearest-neighbor
Coupling time step size	1.0
Fluid time step size	Automatically adjusted (<=1.0)
Solid time step size	1.0

Table 5.5: Coupling setup for the cavity case

The material properties of the fluid and the solid are shown in Table 5.6 and 5.7, respectively.

Property	Symbol	Value
Molecular weight [g/mol]		28.3
Specific heat capacity	c_p	1157.7
Dynamic viscosity	μ	1.84e-05
Prandtl number	Pr	0.7

Table 5.6: Thermophysical properties of the fluid for the cavity case

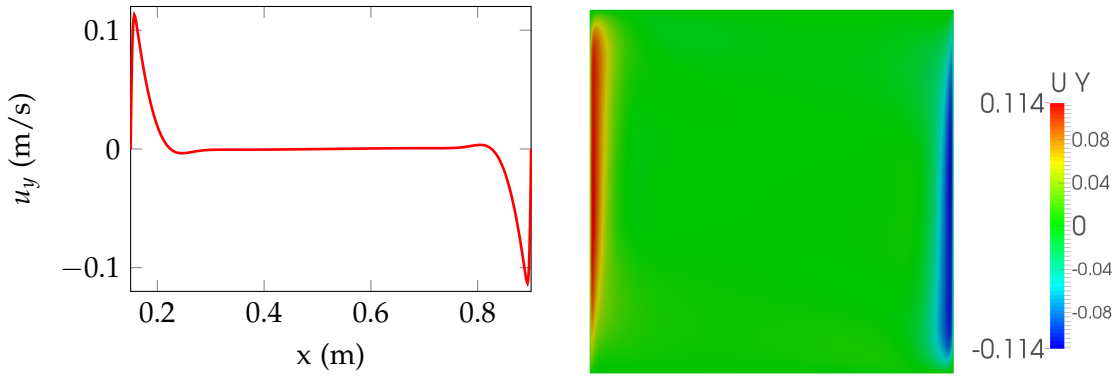
Property	Symbol	Value
Specific heat capacity	c_p	446
Conductivity	k	54
Density	ρ	7800

Table 5.7: Thermophysical properties of the solid for the cavity case

5.2.2 Results

Velocity

In this case with natural convection, the expected behaviour can be observed in the fluid. The warmer fluid moves upwards, and the colder fluid moves downwards, as shown in Figure 5.10a, where the vertical component of the fluid velocity along the horizontal centerline of the cavity is plotted. Figure 5.10b shows the distribution of the vertical component of the velocity.

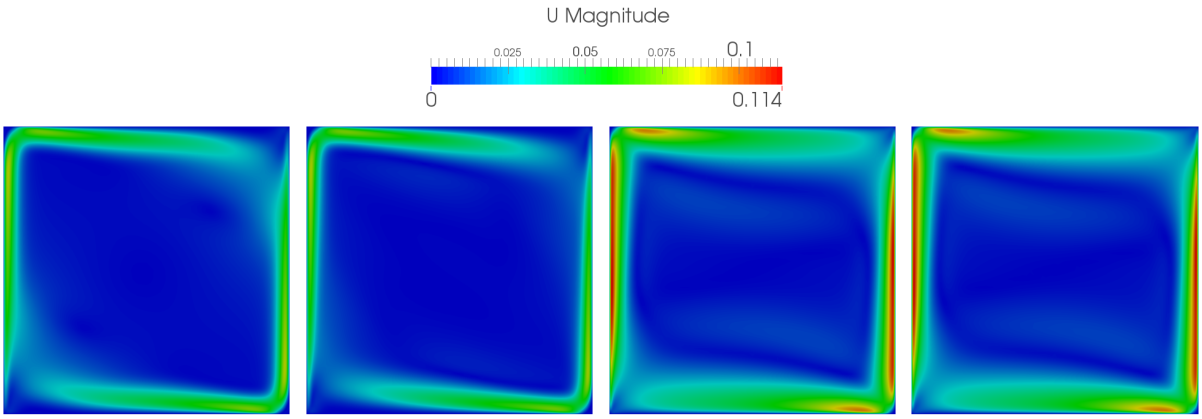


(a) Profile at the horizontal centerline (b) Vertical velocity distribution

Figure 5.10: Vertical component of the velocity at $t = 300$ for the cavity case

In Figure 5.11 the velocity fields at $t = 150$ and $t = 300$ are shown. It can be observed that the temperature difference between the left and right wall sets the fluid into a clock-wise rotating

motion. The results are shown for the coupling using preCICE and for the OpenFOAM CHT solver. The agreement of the results is satisfactory, as no significant differences can be observed.



(a) t=150 (preCICE) (b) t=150 (OpenFOAM) (c) t=300 (preCICE) (d) t=300 (OpenFOAM)

Figure 5.11: Velocity fields for the cavity case

Temperature

Temperature profiles along the horizontal centerline of the cavity are shown in Figure 5.12, for $t = 150$ and $t = 300$. The temperature fields and the isotherms, also at $t = 150$ and $t = 300$, are shown in Figure 5.13. The agreement with the OpenFOAM results is good. Even though in Figure 5.13, for $t = 150$, the isotherms are slightly different, specially in the center part of the cavity, this apparent discrepancy is not considered important: the temperature of the whole center part is very close to 303 K, therefore, even very small deviations make the contours look significantly different.

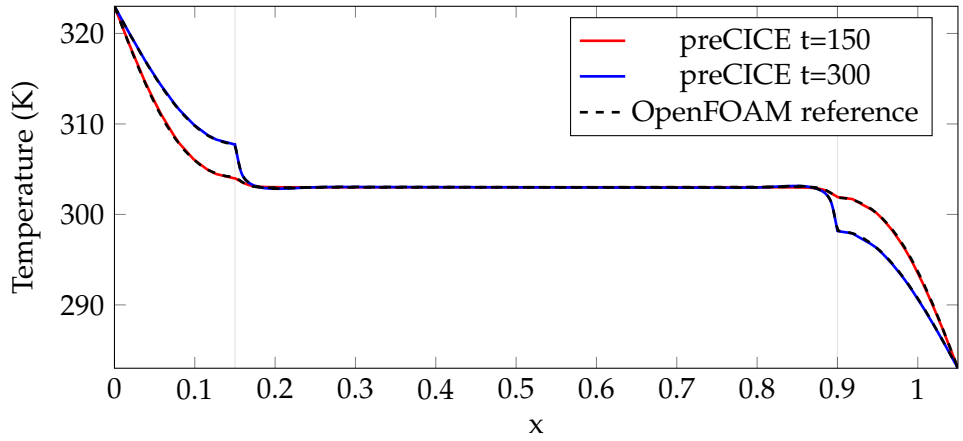
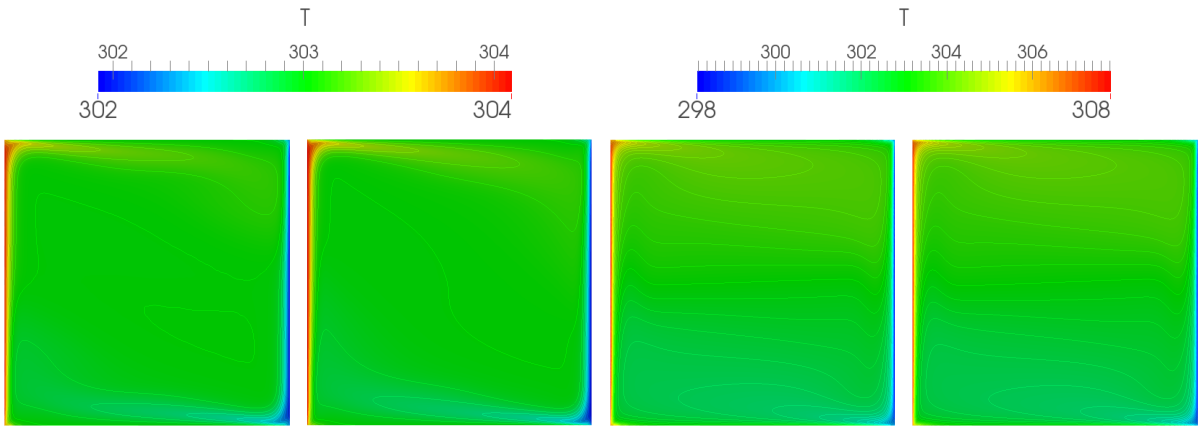


Figure 5.12: Comparison of horizontal temperature profiles for the cavity case (interfaces at $x = 0.15$ and $x = 0.9$)



(a) t=150 (preCICE) (b) t=150 (OpenFOAM) (c) t=300 (preCICE) (d) t=300 (OpenFOAM)

Figure 5.13: Temperature distribution and isotherms in the fluid for the cavity case

Coupling iterations

The number of coupling iterations per time step varied between 1 and 3, with an average of 2.5.

Besides the good agreement between the results obtained using the preCICE coupling and chtMultiRegionFoam, the convergence of the coupling is also good. Therefore, this validation case is considered to have been successfully carried out.

Chapter 6

Demonstration Cases

After having validated the coupling in the previous chapter, this chapter displays some engineering applications where CHT analysis is important, with the aim of demonstrating that the solver developed in this thesis is ready to be used in relevant and complex scenarios.

6.1 Steady-State Simulation of a Shell-and-Tube Heat Exchanger

A heat exchanger is a device that transfers heat between fluids. The shell-and-tube heat exchanger is a particular design of heat exchanger that consists of a cylindrical shell and a bundle of tubes inside the shell. Heat transfer takes place between two fluids that are at different temperatures, one which runs inside the tubes, and another one which runs outside of the tubes, within the shell. Baffles are also often used in the shell-side, in order to direct the flow across the tubes and increase the heat transfer. The fluids do not mix, as they are separated by the solid walls of the tubes. The model used in this thesis is shown in Figure 6.1.

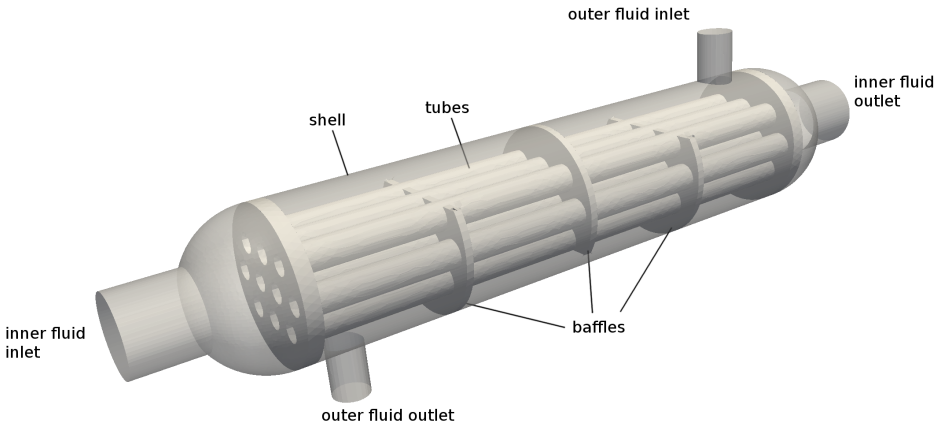


Figure 6.1: Geometry of the simulated shell-and-tube heat exchanger

The purpose of this demonstration case is to show a typical industrial application where CHT

analysis is required. The setup involves a complex geometry consisting of multiple domains, multiple interfaces and participants.

It is interesting to note that the coupled surfaces tend to be relatively large in this type of applications, as these devices are designed to have large contact areas, in order to have a better heat transfer efficiency.

The coupled steady-state solvers were used, given that in this type of applications, the heat transfer and flow characteristics in steady-state are of main interest.

6.1.1 Case Setup

The general setup of the case has been provided by SimScale GmbH (geometry and boundary conditions). It is publicly accessible on the platform[1], where it has been simulated using the solver chtMultiSimpleRegionFoam from OpenFOAM. It has been modified to do the coupling with preCICE, where three participants were used:

1. OpenFOAM – Outer-Fluid (*OF*) (shell-side fluid, running at a higher temperature)
2. OpenFOAM – Inner-Fluid (*IF*) (tube-side fluid, running at a lower temperature)
3. CalculiX/Code_Aster – Solid (*S*)

The coupling dependencies are shown in Figure 6.2. Each of the fluid participants is coupled to the solid participant, but there is no coupling between the fluid participants. Parallel explicit coupling was used.

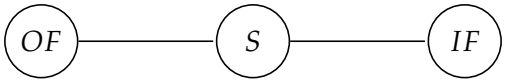


Figure 6.2: Coupling dependencies of the heat exchanger case

The boundary conditions are summarized in Table 6.1. For a better understanding of the geometry of the problem, Figure 6.3 highlights the fluid-solid interfaces. The inner fluid - solid interface, which consists mainly of the inner part of the tubes, is shown in red; and in blue, the outer fluid - solid interface, which consists of the outer part of the tubes and the baffles. The shell of the heat exchanger is not simulated, but assumed to be adiabatic.

Boundary	**Thermal**	**Velocity**
Cold inlet (tube-side)	$T = 283$ K	0.002 m/s
Hot inlet (shell-side)	$T = 353$ K	0.0037 m/s
Tube inner walls	coupled	0 (no slip)
Tube outer walls	coupled	0 (no slip)
Shell walls	adiabatic	0 (no slip)

Table 6.1: Boundary conditions for the heat exchanger case

[1] Available at `https://www.simscale.com/projects/sjesu_rajendra/heat_exchanger-cht_simulation/`. Note that it has been simulated with a different mesh and different material properties.

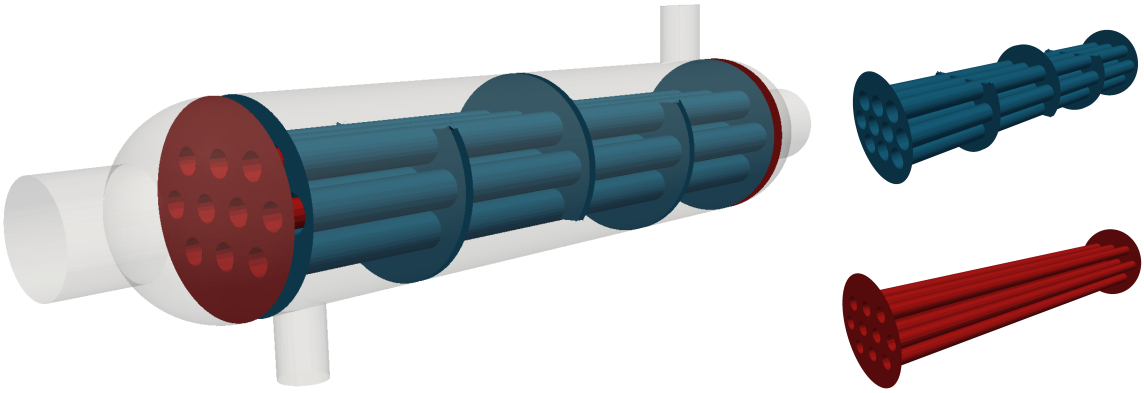


Figure 6.3: Fluid - solid interfaces: inner fluid - solid interface in red, outer fluid - solid interface in blue

The mesh is shown in Figure 6.4. First-order tetrahedral elements were used for the solid mesh. The fluid mesh consists mainly of hexahedral elements, with refinement near to the surfaces. The interfaces are non-matching, and nearest-neighbor was used to map the data between the solid and fluid surface meshes.

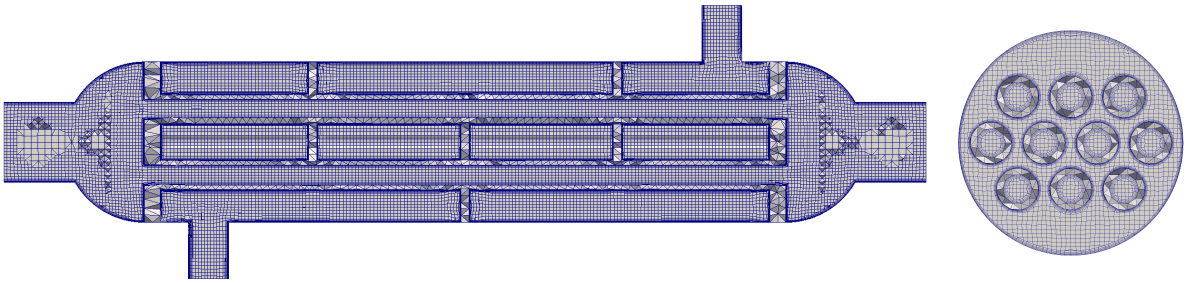


Figure 6.4: Mesh for the heat exchanger case

Participant	**Degrees of freedom**	**Interface size (# of vertices)**
Inner-Fluid	354 444 cells	35 384
Outer-Fluid	573 782 cells	52 809
Solid	41 474 nodes, 127 221 elements	36 788 (interface with Inner-Fluid) 43 836 (interface with Outer-Fluid)

Table 6.2: Discretization of the heat exchanger case

For cases involving complex geometry as the current one, special care must be taken if RBF interpolation is to be used for the mapping of the coupling data. Figure 6.5 shows a close-up to the interface of the solid with the outer (shell-side) fluid, and indicates two places where complications with RBF interpolation might arise. One can see that if the support radius of the radial basis functions is too large, in location 1 of Figure 6.5, points from both sides of the baffle would be used to create the interpolant; similarly, in location 2, points from two

different tubes would be used. A way to overcome this would be to define each individual tube and baffle as a separate surface mesh in preCICE; however, this would make the setup of the case overly complex.

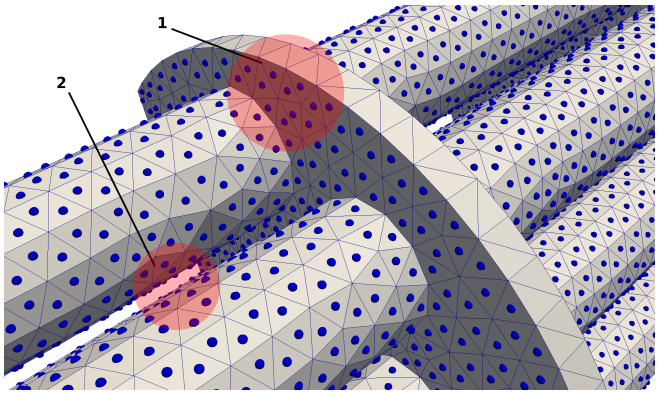


Figure 6.5: Close-up to the interface of the solid with the outer (shell-side) fluid

The thermophysical properties of the fluid and the solid regions can be found in Table 6.3 and 6.4 respectively.

Property	**Symbol**	**Value**
Density	ρ	1027
Specific heat capacity	c_p	4195
Dynamic viscosity	μ	3.645e-4
Prandtl number	Pr	2.289

Table 6.3: Thermophysical properties of the fluid for the heat exchanger case

Property	**Symbol**	**Value**
Specific heat capacity	c_p	385
Conductivity	k	401
Density	ρ	8960

Table 6.4: Thermophysical properties of the solid for the heat exchanger case

6.1.2 Results

The following results are shown for OpenFOAM - CalculiX coupling. The case was also simulated with Code_Aster as solid solver, and also with chtMultiRegionSimpleFoam. No significant differences were observed in the solution fields. The runtimes are compared at the end of this section.

The converged temperature distributions are shown in Figure 6.6. As can be observed, shell-side fluid is cooled down from 353 K to 323 K (measured at the outlet), whereas the tube-side fluid is heated up from 283 K to 293 K.

A rough check of the conservation of energy can be performed by comparing the fluxes at the inlets and outlets and the temperature differences:

$$\begin{aligned}(\rho c_p \Delta(UAT))_{\text{inner fluid}} &= 23.78 \text{ kW} \\ (\rho c_p \Delta(UAT))_{\text{outer fluid}} &= 22.92 \text{ kW}\end{aligned} \tag{6.1}$$

The difference between the energy lost by the warmer fluid and the energy gained by the colder fluid is approximately 3.8%. Considering that the meshes are rather coarse, one could expect that this difference could be reduced by refining the meshes.

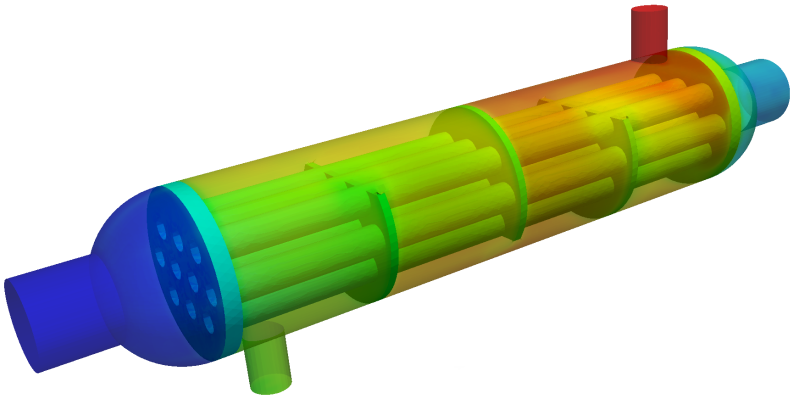
The residual plots for the fluid solvers are shown in Figure 6.7. Given that the adapters only support explicit coupling for steady-state simulations, there is no convergence measurement of the coupling. However, the convergence of the fluid flow fields, as observed in Figure 6.7, implies the convergence of the coupling.

The total runtime was 4190 seconds, for 500 steady-state iterations, with all three participants as serial participants (only interfield parallelism, no intrafield parallelism). Table 6.5 shows the statistics regarding the calls to `initializeData` and `advance`, where the data mapping and data exchange occur. As expected, the percentage of time taken by `advance` is very large for the solid solver, which means that it spends most of the time waiting for the fluid solvers. It turns out, that the Inner-Fluid participant also spent a significant amount of time waiting for the Outer-Fluid participant. Better efficiency should be achieved by assigning more computing cores to the Outer-Fluid participant. Also interesting to note is that the call to `initializeData` takes much longer than an average call to `advance`. This is due to the computation of the mapping between the meshes, which is done once at the beginning, during the first exchange of data. In any case, the time spent by the coupling operations carried out by preCICE are negligible, and the total runtime is determined by the solution time of the most expensive participant, which is in this case the Outer-Fluid participant. The total runtime can be reduced by fine tuning the numerics of the fluid solver, for example, by choosing appropriate under-relaxation factors for the SIMPLE algorithm.

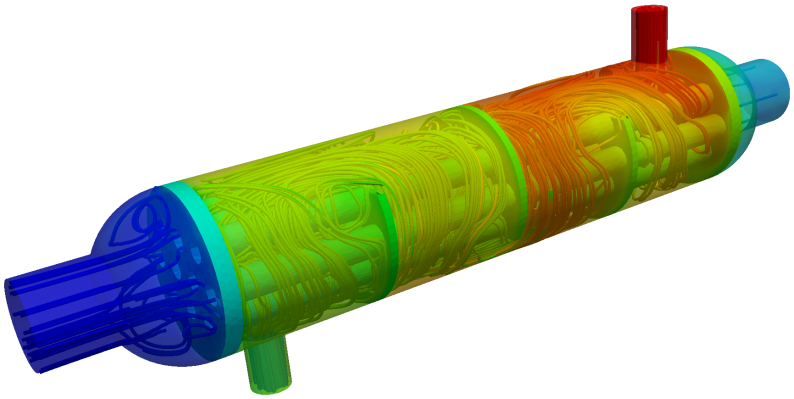
Participant	**Event**	**Count**	**Total (s)**	**Max (s)**	**Min (s)**	**Avg (s)**	**%**
Inner-Fluid	`initializeData`	1	62.2	-	-	-	1.5
Inner-Fluid	`advance`	500	1719.7	70.0	0	3.44	41.0
Outer-Fluid	`initializeData`	1	101.2	-	-	-	2.4
Outer-Fluid	`advance`	500	39.8	30.8	0	0.08	0.9
Solid	`initializeData`	1	142.7	-	-	-	3.4
Solid	`advance`	500	2041.0	19.5	0.03	4.08	48.7

Table 6.5: Statistics of calls to `initializeData` and `advance`, for OpenFOAM-CalculiX coupling, for a total runtime of 4190 s

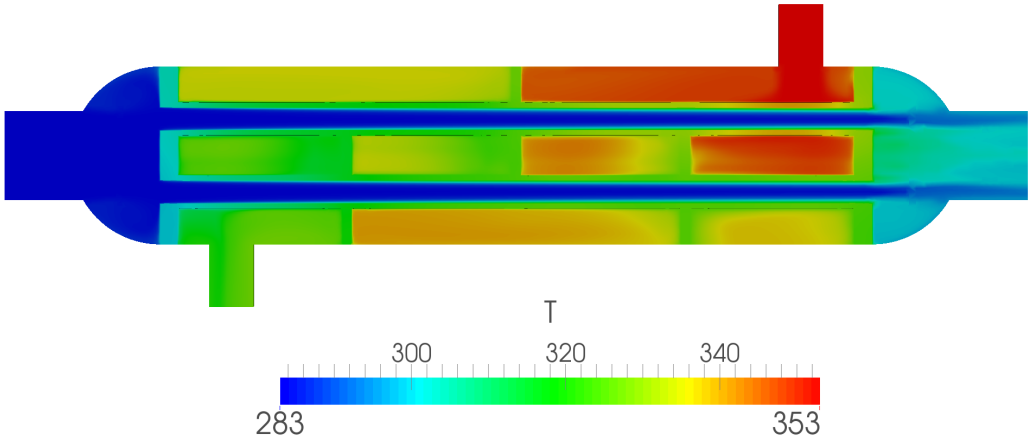
The case was also run using OpenFOAM - Code_Aster coupling. Unlike the flat plate validation case, using nearest-neighbor mapping and first-order elements for the solid did not cause oscillation in the results. The obtained solution fields did not present significant differences



(a) Temperature distribution



(b) Temperature distribution with streamlines



(c) Midplane cut

Figure 6.6: Heat exchanger temperature distribution

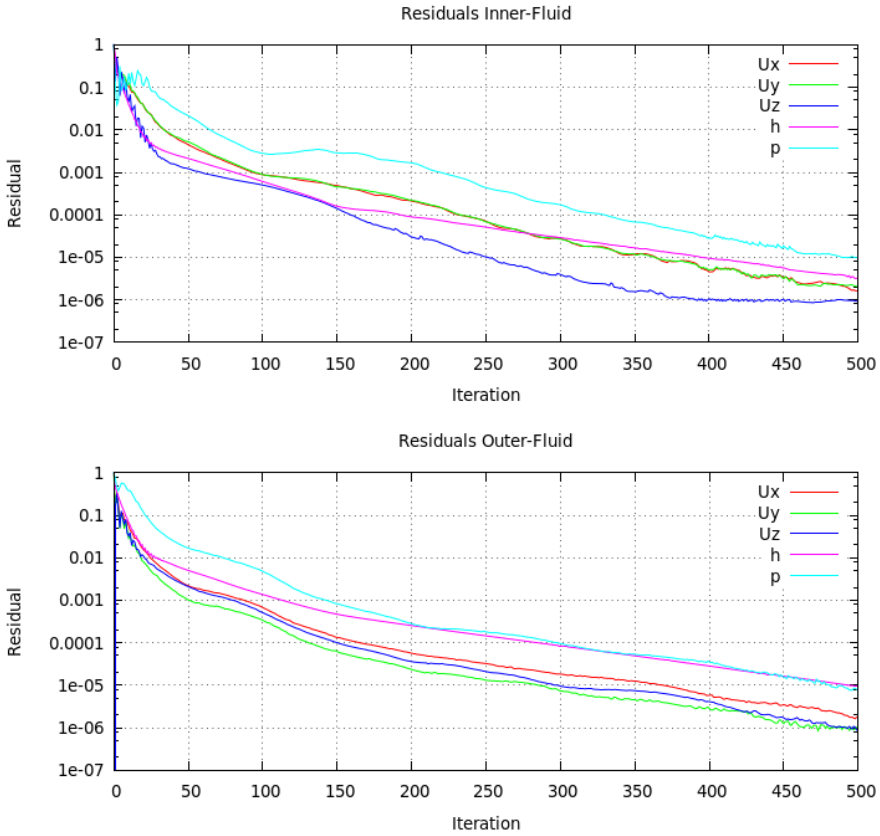


Figure 6.7: Residual plots of the fluid solvers in the heat exchanger simulation

with respect to those obtained using CalculiX as solid solver. The runtime for OpenFOAM - Code_Aster coupling was also similar (4512 s), since it is determined by the fluid solver when using a parallel coupling scheme. There is, however, a small overhead when coupling with Code_Aster, which is the assignment of the boundary conditions during the initialization of the adapter. This operation is carried out element by element, which is not handled efficiently by Code_Aster. In the case of the heat exchanger, this initialization phase additionally required approximately 6 minutes (8% of the total runtime), compared to CalculiX, which basically does not require any start-up time. Future work may try to achieve a more efficient implementation in this respect. The runtime statistics for the coupling with Code_Aster are shown in Table 6.6.

The case was also simulated using chtMultiRegionSimpleFoam. No visible differences were found in the solution fields. The runtimes were also comparable (4625 s with chtMultiRegionSimpleFoam). Nevertheless, the runtime was expected to be significantly lower for the coupling with preCICE, given that interfield parallelism was used, whereas in OpenFOAM, the regions (corresponding to each participant) are solved sequentially. This means that the total runtime should be roughly the sum of the time required to solve each fluid regions. A possible explanation of this unexpected result, is that the case was executed on a dual-core computer with hyperthreading (four threads), such that the interfield parallelism could not be taken full advantage of, as it was not possible to assign a dedicated core to each partici-

Participant	Event	Count	Total (s)	Max (s)	Min (s)	Avg (s)	%
Inner-Fluid	`initialize`	1	363.0	-	-	-	8.0
Inner-Fluid	`initializeData`	1	28.8	-	-	-	0.6
Inner-Fluid	`advance`	500	1480.0	94.4	0	2.96	32.8
Outer-Fluid	`initialize`	1	362.5	-	-	-	8.0
Outer-Fluid	`initializeData`	1	56.4	-	-	-	1.2
Outer-Fluid	`advance`	500	103.0	62.7	0	0.21	2.3
Solid	`initializeData`	1	131.2	-	-	-	2.9
Solid	`advance`	500	1622.7	20.9	0	3.24	36.0

Table 6.6: Statistics of calls to `initializeData` and `advance`, for OpenFOAM-Code_Aster coupling, for a total runtime of 4512 s

pant. However, there is still an advantage over a serial coupling of the participants. A speed up of 1.27 was measured when changing from serial coupling to parallel coupling.

6.2 Simulation of a Pin-Fin Channel Cooling System

Cooling fins are extensions to heat transfer surfaces that are added to enhance the heat transfer rate. They are commonly found in heat sinks. They are also used in the internal cooling channels of turbine blades, as depicted in Figure 6.8. Generally, not only the surface area is important, but also the arrangement of the pins plays a determinant role on the overall efficiency of the cooling system. In this demonstration case, two different arrangements are compared. The pin-fins have a circular cross section, and are inside a rectangular cooling channel. The pins have the same dimensions spacing, and the heat exchange surface areas are the same. This is a simplification of the cooling pin-fins found inside a turbine blade.

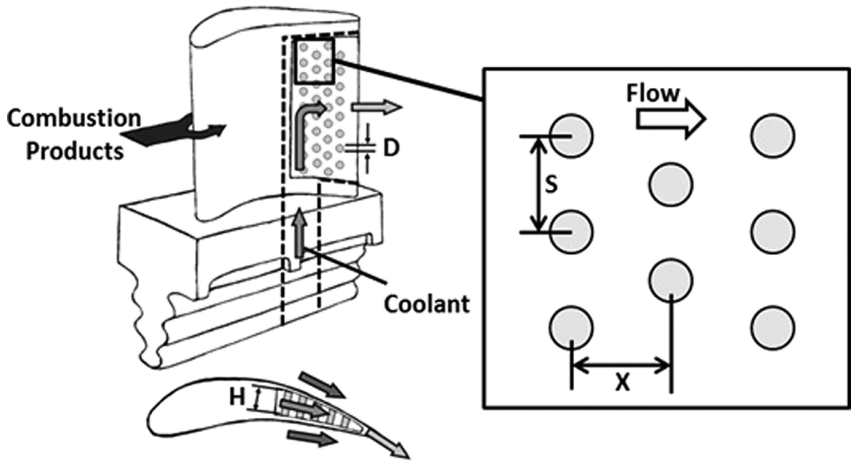


Figure 6.8: Turbine blade with internal cooling on the left, sketch of the pin-fins on the right (source of the image: [22])

The purpose of this demonstration case is to highlight the advantage and importance of doing a coupled analysis, given that even relatively simple geometries can give rise to complex flow patterns that directly affect the heat transfer characteristics of the system.

6.2.1 Case Setup

The computational domain is a rectangular channel. The channel is modelled to be infinitely wide in the spanwise direction by using symmetry boundary conditions. The pins have a circular cross section with a diameter $D = 1.25$ cm. The pins have a streamwise spacing (x) and a spanwise spacing (s) of $x/D = s/D = 1.5$. The channel inner height is 7.5 mm. The thickness of the upper and lower all is 2.5 mm. The length of the simulated portion of the channel is 10 cm. The lower wall of the solid has a constant inward flux of $q_w = 10\ kW/m^2$. The top wall of the solid is adiabatic.

(a) Top view of the fluid domain (b) Front view of the solid domain

Figure 6.9: Geometry of the staggered pin-fin channel

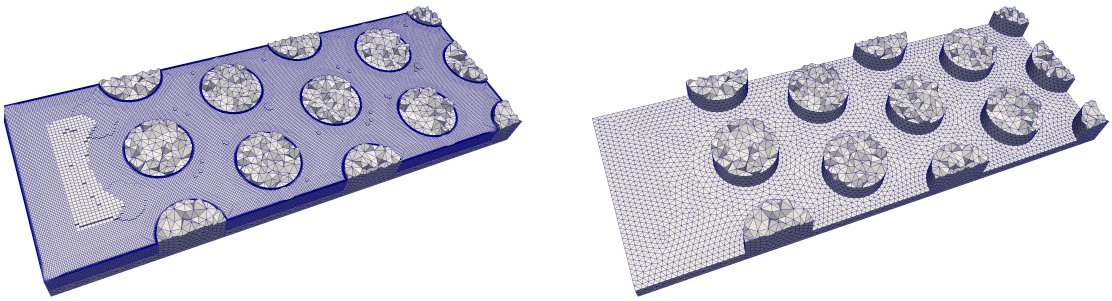
Boundary	**Thermal**	**Velocity**
Solid top wall	adiabatic	-
Solid bottom wall	$q = 10000\ W/m^2$	-
Channel inner walls	coupled	0 (no slip)
Inlet	$T = 300$ K	0.01 m/s

Table 6.7: Boundary conditions for the pin-fin channel

The computational mesh for the staggered arrangement is shown in Figure 6.10 to provide the reader an idea of the element sizes. A similar mesh was used for the in-line arrangement. In this case also first-order tetrahedral elements are used for the solid, and mostly hexahedral elements for the fluid.

The configuration of the coupling is shown in Table 6.9.

Regarding the material properties, for the fluid, the same as Table 6.3 were used. The material properties for the solid are shown in Table 6.10.



(a) Midplane cut of the fluid and solid meshes (b) Midplane cut of the solid mesh

Figure 6.10: Meshes of the staggered pin-fin channel

Participant	Degrees of freedom	Interface size (# of vertices)
Fluid (in-line)	1 433 033 cells	86 334
Fluid (staggered)	1 463 789 cells	86 414
Solid (in-line)	26 442 nodes, 97 688 elements	16 326
Solid (staggered)	25 930 nodes, 96 893 elements	15 824

Table 6.8: Discretization of the pin-fin cooling case

Coupling parameters	
Solvers	OpenFOAM and CalculiX
Coupling boundary conditions	Robin-Robin
Coupling scheme	Parallel-implicit
Coupling convergence acceleration	IQN-ILS (max-used-iterations=100, timesteps-reused=10)
Data mapping	Nearest-neighbor
Coupling time step size	0.01
Fluid time step size	0.01
Solid time step size	0.01

Table 6.9: Coupling setup for the pin-fin cooling case

Property	Symbol	Value
Specific heat capacity	c_p	446
Conductivity	k	54
Density	ρ	8050

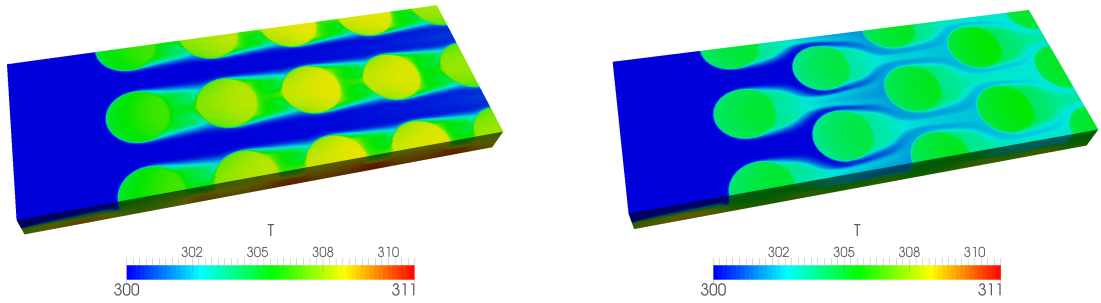
Table 6.10: Thermophysical properties of the solid for the pin-fin cooling case

6.2.2 Results

Steady-State Analysis

In Figure 6.11 the temperature fields in steady-state are shown for each arrangement. It is clear that the temperature distributions are very different for the two arrangements. Table

6.11 compares the temperature values. As can be seen, the temperatures are lower for the staggered arrangement. For this particular setup, the staggered arrangement appears to yield a more efficient cooling.



(a) In-line arrangement (b) Staggered arrangement

Figure 6.11: Temperature at the midplane cut of the pin-fin channel

Temperature	In-line	Staggered
Average solid temperature	307.3 K	304.99 K
Maximum solid temperature	310.753 K	308.52 K
Average interface temperature	307.16 K	304.86 K
Maximum interface temperature	310.54 K	308.165 K

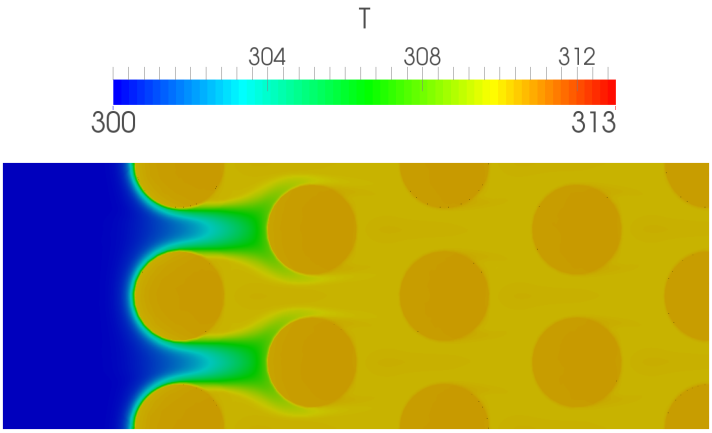
Table 6.11: Pin-fin channel temperature comparison

Transient Analysis

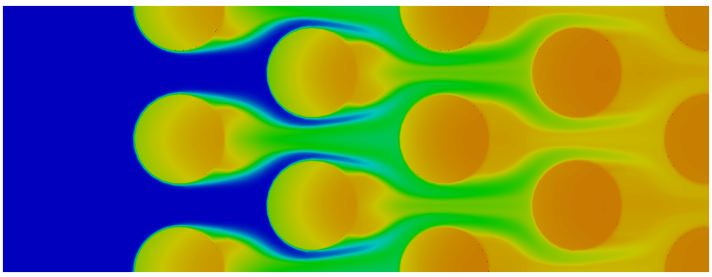
A time step size of 0.01 was used for the transient simulation. Using Robin coupling boundary conditions and IQN-ILS as post-processing, a good convergence was obtained. The first three time steps took 4, 4 and 3 coupling iterations, respectively. Subsequent time steps required only 2 coupling iterations.

Figure 6.12 shows the evolution of the temperature field for the first 6 seconds, at an interval of 2 seconds. The pins are cooled down from left to right. At $t = 4.0$ it is possible to see that the pins on the right side have increased their temperature, because of the constant heat flux from the bottom of the channel. At $t = 6.0$ one can see that the temperature of these pins is already dropping.

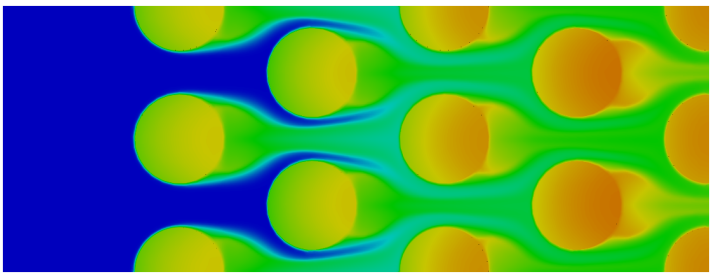
Runtime statistics have been collected for the first 1.5 s of simulation time, and are shown in Table 6.12. The runtime for this period was 28 214 s (7.8 h), on a dual-core computer with one core assigned to each participant. As in the case of the heat exchanger, the solid participant spends a lot of time waiting for the fluid participant. In this case, the difference of the computational costs is even larger, with the solid participant spending 96% of the time in `advance()`, waiting for the fluid participant to synchronize. The total runtime is basically determined by the fluid solver, as the total overhead of the coupling operations does not even add up to 1%.



(a) $t = 2.0$



(b) $t = 4.0$



(c) $t = 6.0$

Figure 6.12: Temperature at the midplane at different times, for the pin-fin cooling case

Participant	**Event**	**Count**	**Total (s)**	**Max (s)**	**Min (s)**	**Avg (s)**	**%**
Fluid	`initializeData`	1	42.3	-	-	-	0.1
Fluid	`advance`	305	12.6	0.1	0	0.04	0.0
Fluid	IQN-ILS-related	-	15.9	-	-	-	0.1
Solid	`initializeData`	1	42.0	-	-	-	0.1
Solid	`advance`	305	27 172.2	114.2	85.2	89.1	96.3

Table 6.12: Runtime statistics for the pin-fin cooling case, for a total runtime of 28 214 s

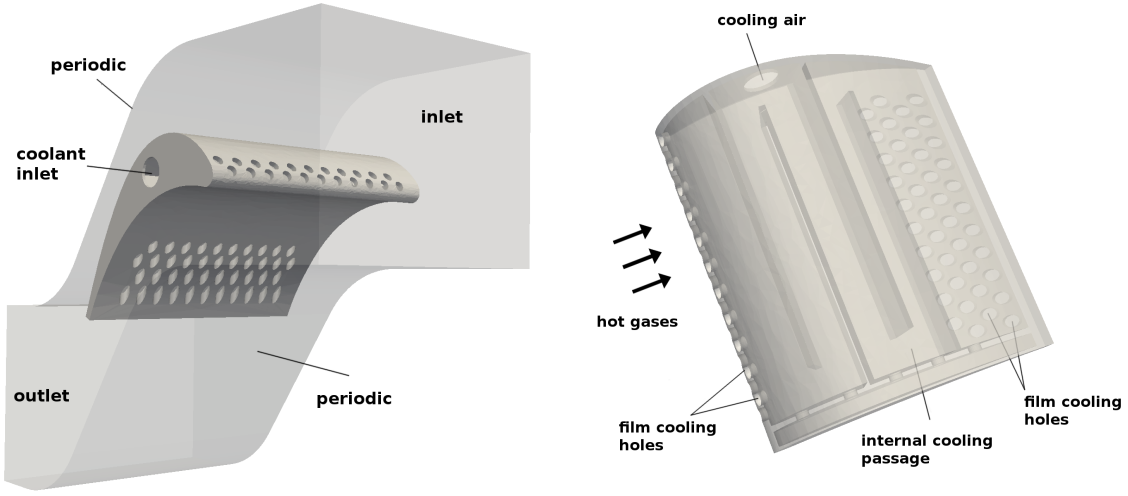
6.3 Steady-State Simulation of Turbine Blade Cooling

Modern turbine blade design aims at increasing the inlet temperature of the turbine. The extremely high inlet temperatures (1300 - 1800 K) often surpass the melting point of the blade materials. Therefore, cooling is essential to protect the blades from damage and thus to increase their lifetime. One very effective cooling technique is film cooling, which consists in injecting a coolant gas through multiple holes of the blade, such that a cooling film is formed around the surface of the blade, protecting it from the high temperatures of the mainstream flow. The holes can be at various locations of the blade, although they are typically located at the leading edge, as it is directly exposed to the high temperatures of the inlet flow.

6.3.1 Case Setup

This demonstration case does not intend to fully reproduce the operating conditions of the turbine blade or to obtain quantitatively accurate results, given that predicting the temperature distributions of such a case in the real world is a highly challenging task, and the allowed uncertainty is very small. Therefore, the purpose is just to verify that the results obtained with the coupling developed in this thesis are qualitatively meaningful for the given setup.

A 3D view of the geometry[2] used for the simulation is shown in Figure 6.13.



(a) Fluid domain and blade geometry

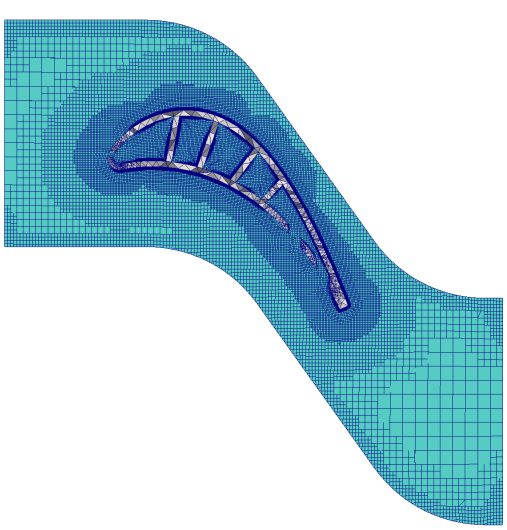
(b) Blade geometry

Figure 6.13: Geometry of the turbine blade cooling case

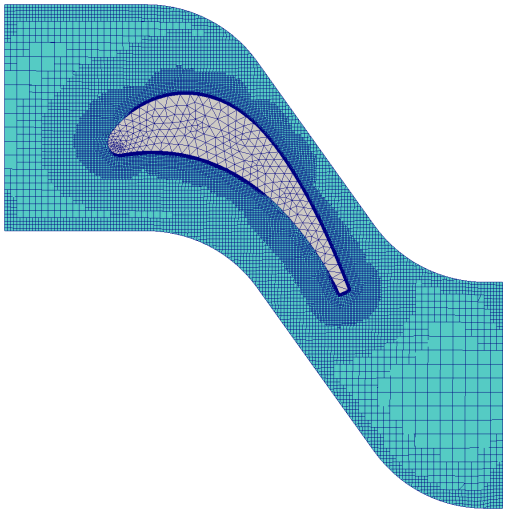
As can be seen in Figure 6.13b, besides the holes for film cooling, the blade also has an internal cooling passage. The fluid and solid meshes are shown in Figure 6.14. Nearest-neighbor mapping was used to map the data between the non-matching meshes. Due to the

[2]The geometry is an adaptation of the "Titanium Turbine Blade" CAD model downloaded from GrabCAD, provided by the user Dan Jewell (`https://grabcad.com/library/titanium-turbine-blade-1`)

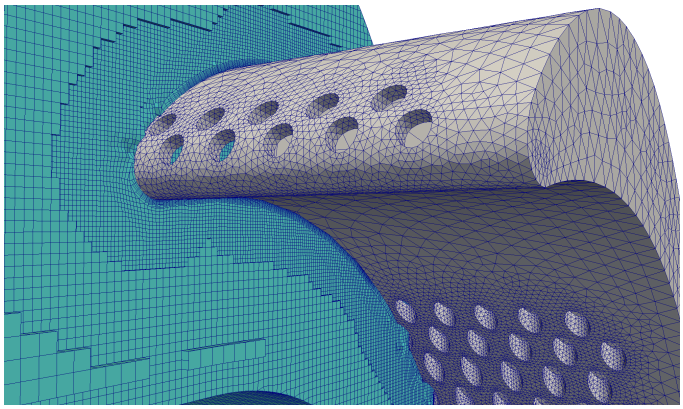
complex geometry involved, choosing an appropriate support radius for RBF interpolation is difficult, as already explained in the heat exchanger case.



(a) Midspan cut of the fluid and solid meshes



(b) Midspan cut of the fluid mesh with the full solid mesh (front view)



(c) Midspan cut of the fluid mesh with the full solid mesh (3D view)

Figure 6.14: Meshes for the turbine blade cooling case

The boundary conditions are listed in Table 6.13. Periodic boundary condition was used for the upper and lower faces of the fluid domain, in order to simulate a vertically stacked array of blades.

The material properties of the fluid are shown in Table 6.15. For the solid, the same as Table 6.10 are used.

Steady-state coupling is used, with a parallel explicit coupling scheme. The case was run on a dual-core computer wit hyperthreading (four threads). The fluid participant uses domain decomposition with two MPI processes, and the solid participant is run serially. The purpose of using this configuration is simply to test that the OpenFOAM adapter works properly when domain decomposition is used.

Boundary	Thermal	Velocity
Inlet	$T_\infty = 788$ K	10 m/s
Coolant inlet	$T_{coolant} = 283$ K	20 m/s
Fluid domain top and bottom walls	periodic	periodic
Fluid domain front and back walls	symmetry	symmetry
Solid domain front and back walls	symmetry	-

Table 6.13: Boundary conditions for the turbine blade cooling case

Participant	Degrees of freedom	Interface size (# of vertices)
Fluid	3 331 745 cells	131 046
Solid	50 035 nodes, 187 046 elements	71 079

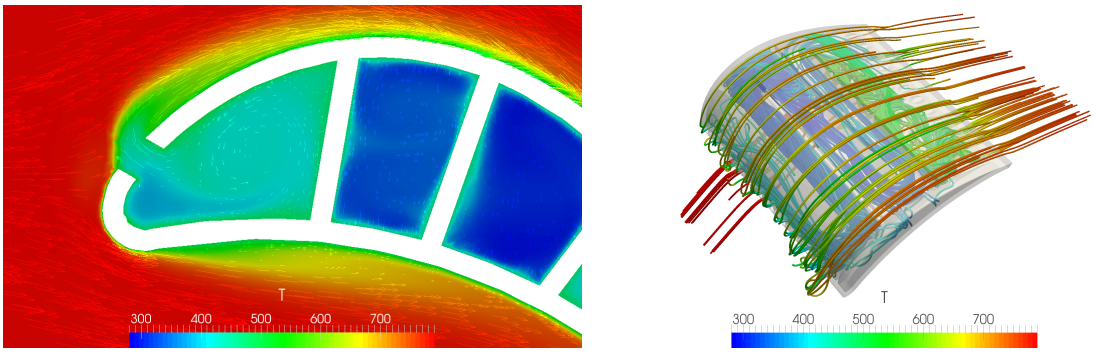
Table 6.14: Discretization of the turbine blade cooling case

Property	Symbol	Value
Molecular weight [g/mol]		28.9
Specific heat capacity	c_p	1007
Dynamic viscosity	μ	1.84e-05
Prandtl number	Pr	0.7

Table 6.15: Thermophysical properties of the fluid for the blade cooling case

6.3.2 Results

Figure 6.15 shows the expected behaviour of the coolant flowing out from the cooling holes and creating a protecting layer around the surface of the blade.



(a) Close up to an injection hole (b) Flow from the internal cooling passages

Figure 6.15: Cooling film around the blade

The performance of the cooling system is typically measured in terms of the overall cooling

efficiency ϕ, which is basically a normalized temperature, defined as:

$$\phi = \frac{T_\infty - T_{blade}}{T_\infty - T_{coolant}} \tag{6.2}$$

In this simulation, ϕ varies between 0.45 and 0.7. The temperature and cooling efficiency distribution are shown in Figure 6.16. The efficiency is highest near to the inlet of the coolant, is intermediate at the leading edge, and is lowest at the trailing edge, at the far side from the coolant inlet. The velocity and pressure distributions are shown in Figure 6.17.

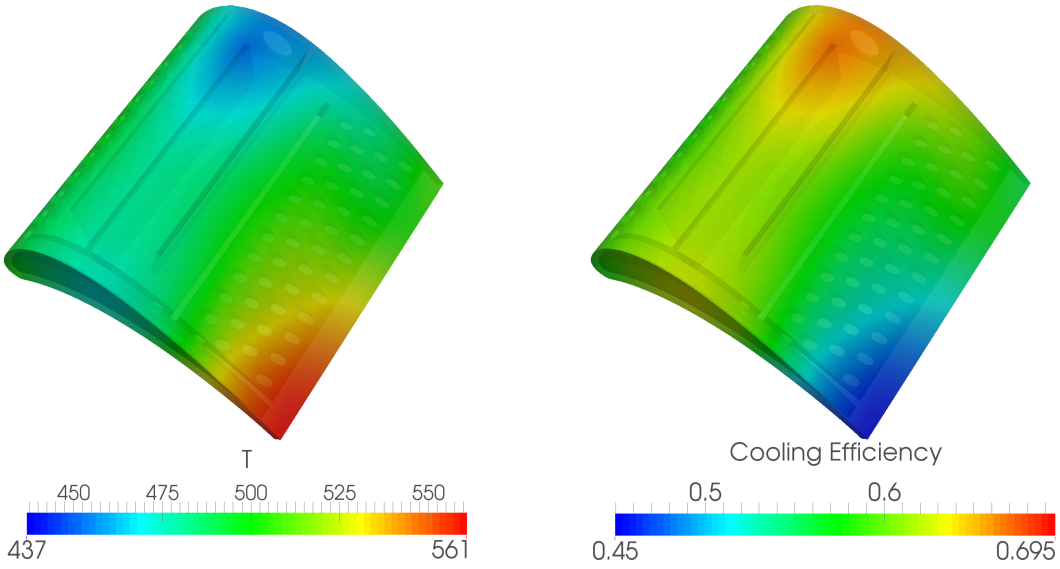
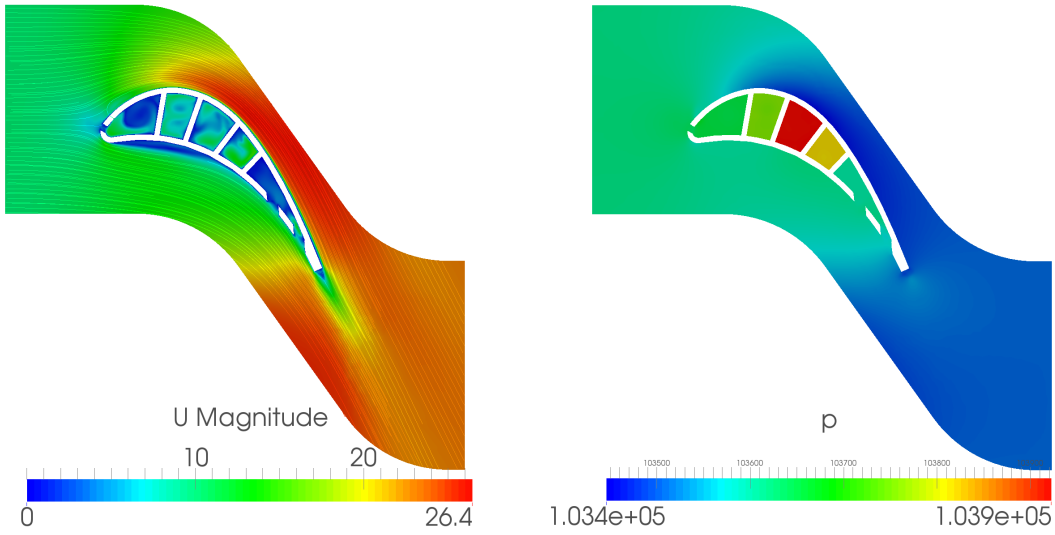


Figure 6.16: Temperature and cooling efficiency distribution on the blade



(a) Velocity distribution with streamlines

(b) Pressure distribution

Figure 6.17: Midspan velocity and pressure distributions for the blade cooling case

A total of 1000 steady-state iterations were carried out. The total runtime was 11.7 hours. Of this total time, the time required by preCICE was only about 10 minutes, most of which corresponding to the initialization phase. Some statistics regarding the time spent by different preCICE operations are shown in Table 6.16.

Participant	Event	Count	Total (s)	Max (s)	Min (s)	Avg (s)	%
Fluid	`advance`	1000	164.4	156.2	0	0.16	0.4
Fluid	`filter mesh`	1	157.6	-	-	-	0.4
Fluid	`initialize`	1	188.2	-	-	-	0.4
Fluid	`initializeData`	1	122.7	-	-	-	0.3
Solid	`advance`	1000	36 400.4	84.5	0	36.4	86.4
Solid	`initialize`	1	13.2	-	-	-	0.0
Solid	`initializeData`	1	529.7	-	-	-	1.3

Table 6.16: Runtime statistics of preCICE events for the turbine blade cooling case with two processes for the fluid participant and one for the solid participant

Chapter 7

Conclusions

A conjugate heat transfer solver was developed by coupling independent fluid and solid solvers, using the coupling library preCICE. Adapters have been developed for OpenFOAM, CalculiX and Code_Aster, and their implementation has been documented. This required the use of three different APIs of preCICE: the C++ API for OpenFOAM, the C API for CalculiX and the Python API for Code_Aster. The adapters can handle different coupling boundary conditions (Dirichlet, Neumann, Robin), multiple interfaces, and they can treat both transient and steady-state problems. The adapters have been validated quantitatively and qualitatively with several test cases, which involve transient and steady-state solutions. This work serves as a basis for future efforts towards integrating preCICE with the SimScale platform, with the aim of adding new multiphysics capabilities to this cloud-based CAE tool.

The flat plate validation case, although simple in terms of setup and geometry, was very useful for identifying what works, what are typical problems and how they can be treated. In this sense, it was found that Robin-Robin coupling results in better stability and convergence than Dirichlet-Neumann coupling. Robin-Robin coupling was stable even with explicit coupling schemes, and was therefore the choice for steady-state coupling, which in the current implementation only supports explicit coupling. The OpenFOAM and CalculiX (transient) adapters allow switching between the different boundary conditions for the coupling, in case that further experiments and comparisons are to be performed. It was also found that the convergence was significantly better when using IQN-ILS compared to Aitken as convergence acceleration scheme. Using Robin-Robin coupling together with IQN-ILS resulted in a good convergence speed, as shown in the validation and demonstration cases where this was used. The main difficulty found in this validation case has been that for some coupling configurations the temperature at the interface could suffer from oscillations which would later lead to the divergence of the solvers. It was found that using a finer solid mesh, using second-order tetrahedral elements for the solid and/or changing the mapping method from nearest-neighbor to RBF interpolation were effective ways to control the oscillations and prevent the solvers from diverging. Identifying robust coupling configurations has been an important step towards the goal of using preCICE on the SimScale platform, given that providing sensible defaults is necessary in order to hide the complexities of the coupling from the users.

The second validation case, consisting of natural convection inside a cavity, was mainly used

to validate the transient solution, by comparing against the results obtained with chtMultiRegionFoam, the transient CHT solver of OpenFOAM. The results of both the flat plate and the cavity validation cases were satisfactory and the adapters are considered to be successfully validated.

In addition to the two validation cases, three demonstration cases were used to test the coupled solvers in more complex scenarios from real industrial applications, which are representative of what users of SimScale would like to simulate on their platform. These cases involve complex geometry and potentially multiple interfaces and multiple participants. The adapters proved to be capable of handling these cases. After having validated the adapters with two cases, the results of the demonstration cases were only qualitatively assessed, and they all showed the expected behaviour. In the case of the heat exchanger, it was simulated using OpenFOAM - CalculiX coupling, OpenFOAM - Code_Aster coupling and chtMultiRegionSimpleFoam (the steady-state CHT solver of OpenFOAM), and no significant differences were found in the solution fields. The runtimes of the demonstration cases have been measured, and it was found that the overhead of the coupling operations are insignificant compared to the total runtime. The runtime measurements also showed that the cases would benefit from using intrafield parallelism for the fluid solver; however, this was not tested in this thesis, and is to be addressed in future work. Nevertheless, there are already other studies, such as [23] and [8], that investigate the scalability of the coupling library preCICE and show that it does not destroy the scalability of the individual solvers.

Although not discussed in the previous chapters, the use of several different tools (the solvers and their pre-/post-processing utilities) can make the setting up of a case somewhat cumbersome. Additional work has to be devoted to making the setup of the case as automatic as possible, in terms of generating the appropriate configuration files, assigning proper boundary conditions for the coupling, setting matching time step sizes if necessary, synchronizing the output frequencies, and so on. For the post-processing, the results of the individual solvers usually have to be converted into another format, so that they can be visualized together. Therefore, besides the challenges regarding stability and convergence of a partitioned approach, there are some other minor drawbacks to running a simulation involving multiple solvers. These also have to be addressed, in order for the integration with the SimScale platform to work.

Overall, the partitioned approach proved to be a viable way of developing a new multiphysics solver, with maximal reuse of existing tools. From the experience of this thesis it was found that the ease of coupling a new solver depends mainly on the interface that is exposed by the solver and whether the solver is intended to be modified and accessed externally. Even though all three solvers are open-source, they differed in the degree of ease with which they could be modified and adapted. Some non-technical difficulties can also slow down the development of the adapters, such as the lack of good documentation of the solvers.

The current work contributes to the repository of the preCICE library with the addition of new coupling adapters, as well as the setups for validation and demonstration cases. Furthermore, the experience acquired in the development of the three coupling adapters can be reused for other multiphysics problems using the same solvers.

Due to time constraints, there are still some limitations in the implemented solution, which may be addressed in future work. Aspects that were not considered in this thesis include, for

example, the modeling of radiation heat transfer and the treatment of non-perfect contacts (i.e. thermal contact resistance). There are also some aspects that require further experimentation and testing. For example, a better understanding of the influence of the location of the coupling data is necessary. It is at the moment not clear whether using different locations for the read- and write-data (e.g. using nodes for read-data and face centers for write-data) can cause oscillations and instabilities. Furthermore, for the solid solvers, the accuracy of the coupling could be potentially be improved by using second-order elements, since they provide more data points (Gauss points) at the interface. In order to take advantage of this, the adapter must be modified to use the individual Gauss points, instead of taking an average between them. With regards to the fluid solver, a better understanding of the checkpointing is necessary, as explained in Section 4.2.2. Finally, more work could also be dedicated to parallelization and scalability testing of the adapters.

Bibliography

[1] Bernhard Gatzhammer. *Efficient and flexible partitioned simulation of fluid-structure interactions*. PhD thesis, Institut für Informatik, Technische Universität München, 2014.

[2] Abram Dorfman and Zachary Renner. Conjugate problems in convective heat transfer: Review. *Mathematical Problems in Engineering*, 2009, 2009.

[3] Tom Verstraete and Sebastian Scholl. Stability analysis of partitioned methods for predicting conjugate heat transfer. *International Journal of Heat and Mass Transfer*, 101:852–869, 2016.

[4] Marc-Paul Errera and Florent Duchaine. Comparative study of coupling coefficients in dirichlet–robin procedure for fluid–structure aerothermal simulations. *Journal of Computational Physics*, 312:218–234, 2016.

[5] Hans-Joachim Bungartz, Florian Lindner, Miriam Mehl, and Benjamin Uekermann. A plug-and-play coupling approach for parallel multi-field simulations. *Computational Mechanics*, 55(6):1119–1129, 2015.

[6] Charbel Farhat and Michael Lesoinne. Two efficient staggered algorithms for the serial and parallel solution of three-dimensional nonlinear transient aeroelastic problems. *Computer methods in applied mechanics and engineering*, 182(3):499–515, 2000.

[7] Joris Degroote, Klaus-Jürgen Bathe, and Jan Vierendeels. Performance of a new partitioned procedure versus a monolithic procedure in fluid–structure interaction. *Computers & Structures*, 87(11):793–801, 2009.

[8] Hans-Joachim Bungartz, Florian Lindner, Miriam Mehl, Klaudius Scheufele, Alexander Shukaev, and Benjamin Uekermann. *Partitioned Fluid–Structure–Acoustics Interaction on Distributed Data: Coupling via preCICE*, pages 239–266. Springer International Publishing, Cham, 2016.

[9] Aukje de Boer, Alexander H van Zuijlen, and Hester Bijl. Comparison of conservative and consistent approaches for the coupling of non-matching meshes. *Computer Methods in Applied Mechanics and Engineering*, 197(49):4284–4297, 2008.

[10] Raad I Issa. Solution of the implicitly discretised fluid flow equations by operator-splitting. *Journal of computational physics*, 62(1):40–65, 1986.

[11] Suhas V Patankar and D Brian Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International journal of heat and mass transfer*, 15(10):1787–1806, 1972.

[12] Henry G Weller, G Tabor, Hrvoje Jasak, and C Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computers in physics*, 12(6):620–631, 1998.

[13] Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische annalen*, 100(1):32–74, 1928.

[14] OpenFOAM User Guide: 4.4 numerical schemes. `http://cfd.direct/openfoam/user-guide/fvschemes/`. Accessed: 2016-11-13.

[15] R Vilums. Implementation of transient robin boundary conditions in openfoam. In *Workshop Multiphysical Modelling in OpenFOAM*, pages 39–40, Riga, Latvia, 2011.

[16] Guido Dhondt. Calculix crunchix user's manual, version 2.7, 2016.

[17] M Vynnycky, S Kimura, K Kanev, and I Pop. Forced convection heat transfer from a flat plate: the conjugate problem. *International Journal of Heat and Mass Transfer*, 41(1):45–59, 1998.

[18] F Ampofo and TG Karayiannis. Experimental benchmark data for turbulent natural convection in an air filled square cavity. *International Journal of Heat and Mass Transfer*, 46(19):3551–3572, 2003.

[19] G Barakos, E Mitsoulis, and Do Assimacopoulos. Natural convection flow in a square cavity revisited: laminar and turbulent models with wall functions. *International Journal for Numerical Methods in Fluids*, 18(7):695–719, 1994.

[20] Mikhail A Sheremet and Igor V Miroshnichenko. Numerical study of turbulent natural convection in a cube having finite thickness heat-conducting walls. *Heat and Mass Transfer*, 51(11):1559–1569, 2015.

[21] Anil Kumar Sharma, K Velusamy, and C Balaji. Turbulent natural convection of sodium in a cylindrical enclosure with multiple internal heat sources: A conjugate heat transfer study. *International Journal of Heat and Mass Transfer*, 52(11):2858–2870, 2009.

[22] Jason K Ostanek. Improving pin-fin heat transfer predictions using artificial neural networks. *Journal of Turbomachinery*, 136(5):051010, 2014.

[23] Benjamin Uekermann. *Partitioned Fluid-Structure Interaction on Massively Parallel Systems*. PhD thesis, Institut für Informatik, Technische Universität München, 2016.